Version 2.0

DEPARTMENT OF THE AIR FORCE
Software Technology Support Center

GUIDELINES for SUCCESSFUL ACQUISITION and MANAGEMENT of SOFTWARE-INTENSIVE SYSTEMS:

Weapon Systems
Command and Control Systems
Management Information Systems

JUNE 1996

Volume 1

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19961004 060

Version 2.0

DEPARTMENT OF THE AIR FORCE
Software Technology Support Center

GUIDELINES for SUCCESSFUL ACQUISITION and MANAGEMENT of SOFTWARE-INTENSIVE SYSTEMS:

Weapon Systems
Command and Control Systems
Management Information Systems

JUNE 1996

Volume 1

Blank page.

Preface

Blank page.

PREFACE

Every time these Guidelines go to press [this is our third version], we like to reflect on where the software community was when we started this project. With this publication, we again realize it marks another significant milestone in improving how we acquire and manage our major software-intensive systems. Indeed, we have come a long way from the speech I made in 1990 to a gathering of software professionals. At that time I declared that the 1980s were a lost decade from the perspective of software development progress. The question I posed was: "*Will there be a breakthrough in the 1990s?*" I went on to say: "*It won't happen automatically; people are too satisfied with our unsatisfactory ways. We dare not make the mistake of complacency a la the automobile industry; we must push awareness and resource commitment to get ahead of the power curve of demand. Some believe that we only tap about 5% of the information that will be tapped and exploited by 2010! Demands on systems, especially management information systems, will be overwhelming; we must be able to feed and satisfy this monster or it will devour us. The challenge, then, is how to make the software world of the year 2000 a better place to be. If we succeed, the 1990s will be known as a banner decade for software. Should we fail, however, we will be hindered by the continuance of the software crisis and of the craft technologies of the 1970s and 1980s, even if we implement their products on gigahertz processors. Worse, because such methods of developing software are so costly and unreliable, there will be few resources left over for the technology transition task, and new technologies and techniques will lag unaided into practice at the same glacial rate they do today.*"

In 1994, I closed the annual Software Technology Conference at Salt Lake City, Utah with the observation that the underlying need within the defense community is for ***predictability***. "*From a Pentagon perspective, it is not the fact that software costs are growing annually and consuming more and more of our defense dollars that worries us. Nor is it the fact that our weapon systems and commanders are becoming more and more reliant on software to perform their mission. Our inability to predict how much a software system will cost, when it will be operational, and whether or not it will satisfy user requirements, is the major concern. What our senior managers and DoD leaders want most from us, is to deliver on our promises. They want systems that are on-time, within budget, that satisfy user requirements, and are reliable.*" I told the audience that the these Guidelines represent the most comprehensive source document on how to achieve these goals. They are now required reading by every major defense

university, used by industry as they prepare for competitive software procurements, and widely followed by software engineers in the field, the private sector, and among the services. With this publication, they continue to represent the most substantive compilation of lessons-learned and best practices gathered from recognized software practitioners and experts available anywhere — government and industry-wide.

So, as we go to press, "*Where are we today and where will we be tomorrow?*" Although we have come a long way, as a reminder that we have not arrived, we are reprinting as the Foreword to this document the September 1994 *Scientific American* article entitled, "Software's Chronic Crisis," by Wayt Gibbs. We have not arrived, but we have largely moved, at least conceptually, beyond the day when entrepreneurs (i.e., brilliant programmers) were relied upon to develop and deliver satisfactory software systems. At the midpoint of the 1990s, software engineering and associated elements of software process maturity have come to be more the norm. There are many good Level 3 contractors. In our acquisitions we are now looking, not just to see if a contractor is a Level 3, but also at how close he is to a Level 4! Contractors and program offices are generating metrics and using them for management. We now have the means to progress into an era of predictable development of large-scale software-intensive systems.

Back in 1990, I said that if the 1990s reveal a *silver bullet,* that bullet will be ***reuse***. That was perhaps naive. Reuse, as a concept, has been discussed and cussed almost continually since then; and yet, it is still not practiced on any major scale. In retrospect, the practice of software engineering discipline and process maturity discussed here probably were prerequisites. Has the time now come? Will major benefits from reuse yet arise before the end of this decade? I believe the potential is very real. The question is whether you, as program managers, move toward it with purpose, as in *Where are we going?* or with malaise, as in *Whither are we drifting?*

Reuse is a software technology term. I am convinced that software technology alone, will not bring any more reuse than we are seeing today. To promote reuse, we must think in terms of *architecture-based product-lines.* In these Guidelines you will learn about DoD's Technical Architecture for Information Management (TAFIM), and more specifically *building codes,* which are the prerequisites for *architecture.* The Air Force experience with a Cleanroom-centered product-line at Space Command, established under the STARS program, has shown spectacular results. Productivity has increased from 175 lines-of-code (LOC) per month to 1,875 LOC per month. Defects have decreased from 3+ defects per KLOC to 0.35 defects per KLOC, and costs have been reduced from $130 to about $50 per LOC. Perhaps even more important is the reduction in cycle time. A new application was recently completed in approximately 6 months. This application involved the production of 240,000 LOC, of which 60% were from reuse, 30% were CASE tool generated, and 10% were handwritten. The Army and Navy STARS product-

Preface

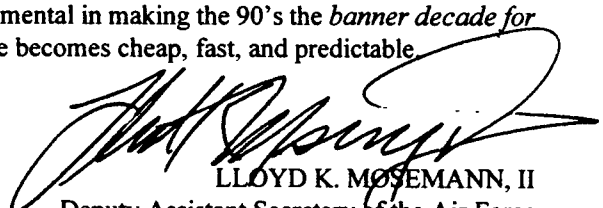
lines have demonstrated similar results.

Product-lines go beyond software technology in that management must understand the wisdom of making capital investment in the product-line — hiring an architect, architecture creation, reusable component development and pre-certification of commercial-off-the-shelf (COTS) software components. Both government and industry must move away from paradigms which focus on specific requirements and funding/management which is single system oriented. As product-lines are established and evolve, I believe they will gather momentum quickly. The contractor with an effective product-line will win competitions "*hands down*" because of lower cost, higher quality, quicker delivery, and above all, their predictable performance track record.

Our reflections are not complete without a word about Ada. Ada has arrived. Even as DoD moves from mandating Ada to preferring Ada (the first choice is good, existing COTS), any company would be foolish to establish a product-line based on any other language now known. The special features of Ada, such as tasking and exception handling, make it mandatory for any application involving safety of life; hence, Boeing's choice for its fly-by-wire 777, and the almost unanimous choice for running European railways. Ada was designed with reuse in mind. Ada was an object-oriented language before the term *object-oriented* was even coined. Why else were Ada projects chosen *best in show* at the last three *Object Worlds?* Even without a product-line, 30% reuse is the norm when developing Ada programs with a mature software engineering environment, such as Rational's Apex.

COTS is the stated DoD desire, and COTS is the epitome of product-line reuse. We need COTS; but, the answer is to write COTS in Ada. Smart companies, whether currently in the COTS business, or considering entering the COTS world in the context of architecture-based product-line establishment, will make independent, and objective, evaluation of Ada against the alternatives, with the same zeal that they make cost trade-off studies when considering a new plant site or other capital investment.

By the time you read these Guidelines, I will have retired from my position as Deputy Assistant Secretary of the Air Force for Communications, Computers, and Support Systems. I feel confident we will make the strides during the last half of the 1990s that we have made in the first half; and the guidance found in this book is a positive step in that direction. Read, digest, and act upon the sound management principles and methodologies discussed here, and you will be instrumental in making the 90's the *banner decade for software* — where software becomes cheap, fast, and predictable.



LLOYD K. MOSEMANN, II
Deputy Assistant Secretary of the Air Force
(Communications, Computers, & Support Systems)

Blank page.

Foreword

Blank page.

FOREWORD

Software's Chronic Crisis

Originally published in *Scientific American* magazine, September 1994. *[Reprinted with Permission. Copyright ©1994 by Scientific American, Inc. All rights reserved.]*

W. Wayt Gibbs
Scientific American Staff Writer

Denver's new international airport was to be the pride of the Rockies, a wonder of modern engineering. Twice the size of Manhattan, 10 times the breadth of Heathrow, the airport is big enough to land three jets simultaneously — in bad weather. Even more impressive than its girth is the airport's subterranean baggage-handling system. Tearing like intelligent coal-mine cars along 21 miles of steel track, 4,000 independent "*telecars*" route and deliver luggage between the counters, gates, and claim areas of 20 different airlines. A central nervous system of some 100 computers networked to one another and to 5,000 electric eyes, 400 radio receivers, and 56 bar-code scanners orchestrates the safe and timely arrival of every valise and ski bag.

At least that is the plan. For nine months, this Gulliver has been held captive by Lilliputians — errors in the software that controls its automated baggage system. Scheduled for takeoff by last Halloween, the airport's grand opening was postponed until December to allow BAE Automated Systems time to flush the gremlins out of its $193-million system. December yielded to March. March slipped to May. In June the airport's planners, their bond rating demoted to junk and their budget hemorrhaging red ink at the rate of $1.1 million a day in interest and operating costs, conceded that they could not predict when the baggage system would stabilize enough for the airport to open.

Software glitches in an automated baggage-handling system force Denver International Airport to sit empty nine months after airplanes were to fill gates and runways. The system that is supposed to shunt luggage in 4,000 independent "*telecars*" along 21 miles of track still opened, damaged, and misrouted cargo as testing continued in July.

FOREWORD *"Software's Chronic Crisis"*

To veteran software developers, the Denver debacle is notable only for its visibility. Studies have shown that for every six new large-scale software systems that are put into operation, two others are cancelled. The average software development project overshoots its schedule by half; larger projects generally do worse. And some three quarters of all large systems are "*operating failures*" that either do not function as intended or are not used at all.

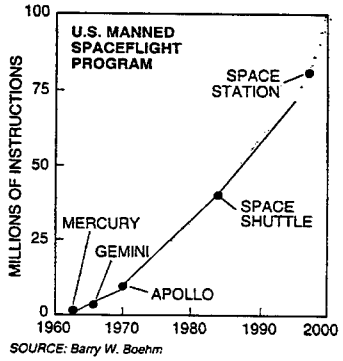
The art of programming has taken 50 years of continual refinement to reach this stage. By the time it reached 25, the difficulties of building big software loomed so large that in the autumn of 1968 the NATO Science Committee convened some 50 top programmers, computer scientists, and captains of industry to plot a course out of what had come to be known as the software crisis. Although the experts could not contrive a road map to guide the industry toward firmer ground, they did coin a name for that distant goal: software engineering, now defined formally as "*the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.*"

A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is stall handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently. *"It's like musket making was before Eli Whitney,"* says Brad J. Cox, a professor at George Mason University. *"Before the industrial revolution, there was a nonspecialized approach to manufacturing goods that involved very little interchangeability and a maximum of craftsmanship. If we are ever going to lick this software crisis, we're going to have to stop this hand-to-mouth, every-programmer-builds-everything-from-the-ground-up preindustrial approach."*

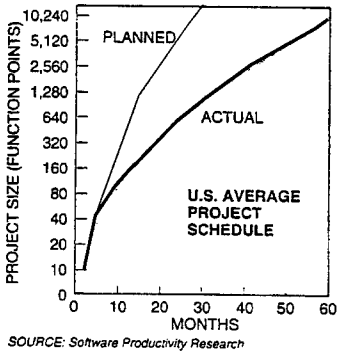
The picture is not entirely bleak. Intuition is slowly yielding to analysis as programmers begin using quantitative measurements of the quality of the software they produce to improve the way they produce it. The mathematical foundations of programming are solidifying as researchers work on ways of expressing program designs in algebraic forms that make it easier to avoid serious mistakes. Academic computer scientists are starting to address their failure to produce a solid corps of software professionals. Perhaps, most important, many in the industry are turning their attention toward inventing the technology and market structures needed to support interchangeable, reusable software parts.

"Unfortunately, the industry does not uniformly apply that which is well-known best practice," laments Larry E. Druffel, director of Carnegie Mellon University's Software Engineering Institute. In fact, a research innovation typically requires 18 years to wend its way into the repertoire of standard programming techniques. By combining their efforts, academia, industry, and government may be able to hoist software development to the level of an industrial-age engineering discipline within the decade. If they

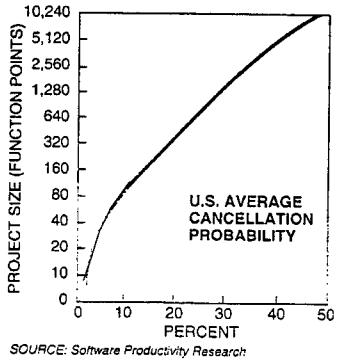
come up short, society's headlong rush into the information age will be halting and unpredictable at best.



SOURCE: Barry W. Boehm



SOURCE: Software Productivity Research



SOURCE: Software Productivity Research

Software is exploding in size as society comes to rely on more powerful computer systems *(top)*. **That faith is often rewarded by disappointment, as most large software projects overrun their schedules** *(middle)* **and many fail outright** *(bottom)* **— usually after most of the development money has been spent.**

FOREWORD *"Software's Chronic Crisis"*

Despite 50 years of progress, the software industry remains years — perhaps decades — short of the mature engineering discipline needed to meet the demands of an information-age society.

SHIFTING SANDS

"We will see massive changes [in computer use] *over the next few years, causing the initial personal computer revolution to pale into comparative insignificance,"* concluded 22 leaders in software development from academia, industry, and research laboratories this past April. The experts gathered at Hedsor Park, a corporate retreat near London, to commemorate the NATO conference and to analyze the future directions of software. *"In 1968 we knew what we wanted to build but couldn't,"* reflected Cliff Jones, a professor at the University of Manchester. *"Today we are standing on shifting sands."*

The foundations of traditional programming practices are eroding swiftly, as hardware engineers churn out ever faster, cheaper, and smaller machines. Many fundamental assumptions that programmers make — for instance, their acceptance that everything they produce will have defects — must change in response. *"When computers are embedded in light switches, you've got to get the software right the first time because you're not going to have a chance to update it,"* says Mary M. Shaw, a professor at Carneige-Mellon.

"The amount of code in most consumer products is doubling every two years," notes Remi H. Bourgonjon, director of software technology at Philips Research Laboratory in Eindhoven. Already, he reports, televisions may contain up to 500 kilobytes of software; an electric shaver, two kilobytes. The power trains in new General Motors cars run 30,000 lines of computer code.

Getting software right the first time is hard even for those who care to try. The Department of Defense applies rigorous — and expensive — testing standards to ensure that software on which a mission depends is reliable. Those standards were used to certify Clementine, a satellite that the DoD and the National Aeronautics and Space Administration directed into lunar orbit this past spring. A major part of the Clementine mission was to test targeting software that could one day be used in a space-based missile defense system. But when the satellite was spun around and instructed to fix the moon in its sights, a bug in its program caused the spacecraft instead to fire its maneuvering thrusters continuously for 11 minutes. Out of fuel and spinning wildly, the satellite could not make its rendezvous with the asteroid Geographos.

Errors in real-time systems such as Clementine are devilishly difficult to spot because, like that suspicious sound in your car engine, they often

occur only when conditions are just so [see "The Risks of Software," by Bev Littlewood and Lorenzo Strigini; *Scientific American*, November 1992]. *"It is not clear that the methods that are currently used for producing safety-critical software, such as that in nuclear reactors or in cars, will evolve and scale up adequately to match our future expectations,"* warned Gilles Kahn, the scientific director of France's INRIA research laboratory, at the Henson Park meeting. *"On the contrary, for real-time systems I think we are at a fracture point."*

Software is buckling as well under tectonic stresses imposed by the inexorably growing demand for *"distributed systems:"* programs that run cooperatively on many networked computers. Businesses are pouring capital into distributed information systems that they hope to wield as strategic weapons. The inconstancy of software development can turn such projects into Russian roulette.

Many companies are lured by goals that seem simple enough. Some try to reincarnate obsolete mainframe-based software in distributed form. Others want to plug their existing systems into one another or into new systems with which they can share data and a friendlier user interface. In the technical lingo, connecting programs in this way is often called systems integration. But Brian Randell, a computer scientist at the University of Newcastle upon Tyne, suggests that *"there is a better word than integration, from old R.A.F. slang: namely, 'to graunch,' which means 'to make to fit by the use of excessive force.'"*

It is a risky business, for although software seems like malleable stuff, most programs are actually intricate plexuses of brittle logic through which data of only the right kind may pass. Like handmade muskets, several programs may perform similar functions and yet still be unique in design. That makes software difficult to modify and repair. It also means that attempts to graunch systems together often end badly. In 1987, for example, California's Department of Motor Vehicles decided to make its customer's lives easier by merging the state's driver and vehicle registration systems — a seemingly straightforward task. It had hoped to unveil convenient one-stop renewal kiosks last year. Instead the DMV saw the projected cost explode to 6.5 times the expected price and the delivery date recede to 1998. In December the agency pulled the plug and walked away from the seven-year, $44.3-million investment.

Sometimes nothings fails like success. In the 1970s American Airlines constructed SABRE, a virtuosic, $2-billion flight reservation system that became part of the travel industry's infrastructure. *"SABRE was the shining example of a strategic information system because it drove American to being the world's largest airline,"* recalls Bill Curtis, a consultant to the Software Engineering Institute.

Intent on brandishing software as effectively in this decade, American tried to graunch its flight-booking technology with the hotel and car reservation

systems of Marriott, Hilton, and Budget. In 1992 the project collapsed into a heap of litigation. *"It was a smashing failure,"* Curtis says. *"American wrote off $165 million against that system."*

The airline is hardly suffering alone. In June IBM's Consulting Group released the results of a survey of 24 leading companies that had developed large distributed systems. The numbers were unsettling: 55% of the projects cost more than expected, 68% overran their schedules, and 88% had to be substantially redesigned.

The survey did not report one critical statistic: how reliably the completed programs ran. Often systems crash because they fail to expect the unexpected. Networks amplify this problem. *"Distributed systems can consist of a great set of interconnected single points of failure, many of which you have not identified beforehand,"* Randell explains. *"The complexity and fragility of these systems pose a major challenge."*

The challenge of complexity is not only large but also growing. The bang that computers deliver per buck is doubling every 18 months or so. One result *is "an order of magnitude growth in system size every decade — for some industries, every half decade,"* Curtis says. To keep up with such demand, programmers will have to change they way that they work. *"You can't build skyscrapers using carpenters,"* Curtis quips.

MAYDAY, MAYDAY

When a system becomes so complex that no one manager can comprehend the entirety, traditional development processes break down. The Federal Aviation Administration (FAA) has faced this problem throughout its decade-old attempt to replace the nation's increasingly obsolete air-traffic control system [see "Aging Airways," by Gary Stix; *Scientific American*, May 1994].

The replacement, called the Advanced Automation System (AAS), combines all the challenges of computing in the 1990s. A program that is more than a million lines in size is distributed across hundreds of computers and embedded into new and sophisticated hardware, all of which must respond around the clock to unpredictable real-time events. Even a small glitch potentially threatens public safety.

To realize its technological dream, the FAA chose IBM's Federal Systems Company, a well-respected leader in software development that has since been purchased by Loral. FAA managers expected (but did not demand) that IBM would use state-of-the-art techniques to estimate the cost and length of the project. They assumed that IBM would screen the requirements and design drawn up for the system in order to catch mistakes early, when they can be fixed in hours rather than days. And the FAA conservatively expected to

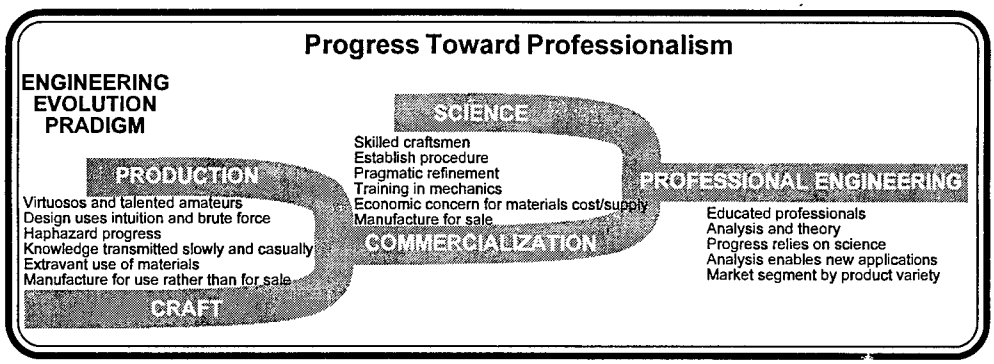
pay about $500 per line of computer code, five times the industry average for well-managed development processes.

According to the report on the AAS project released in May by the Center for Naval Analysis, IBM's *"cost estimation and development process tracking used inappropriate data, were performed inconsistently, and were routinely ignored"* by project managers. As a result, the FAA has been paying $700 to $900 per line for the AAS software. One reason for the exorbitant price is that *"on average every line of code developed needs to be rewritten once,"* bemoaned an internal FAA report.

Alarmed by skyrocketing costs and tests that showed the half-completed system to be unreliable, FAA Administrator David R. Hinson decided in June to cancel two of the four major parts of the AAS and to scale a third. The $144 million spent on these failed programs is but a drop next to the $1.4 billion invested in the fourth and central piece: new workstation software for air-traffic controllers.

That project is also spiraling down the drain. Now running about five years late and more than $1 billion over budget, the bug-infested program is being scoured by software experts at Carnegie-Mellon and the Massachusetts Institute of Technology to determine whether it can be salvaged or must be canceled outright. The reviewers are scheduled to make their report in September.

Disaster will become an increasingly common and disruptive part of software development unless programming takes on more of the characteristics of an engineering discipline rooted firmly in science and mathematics *[see following boxes].* Fortunately, that trend has already begun. Over the past decade industry leaders have made significant progress toward understanding how to measure, consistently and quantitatively, the chaos of their development processes, the density of errors in their products, and the stagnation of their programmers' productivity. Researchers are already taking the next step: finding practical, repeatable solutions to these problems.



FOREWORD *"Software's Chronic Crisis"*

Engineering disciplines share common stages in their Evolution, observes Mary M. Shaw of Carnegie-Mellon University. She spies interesting parallels between software engineering and chemical engineering, two fields that aspire to exploit, on an industrial scale, the processes that are discovered by small-scale research.

Like software developers, chemical engineers try to design processes to create safe, pure products as cheaply and quickly as possible. Unlike most programmers, however, chemical engineers rely heavily on scientific theory, mathematical modeling, proven design solutions, and rigorous quality-control methods — and their efforts usually succeed.

Software, Shaw points out, is somewhat less mature, more like a cottage industry than a professional engineering discipline. Although the demand for more sophisticated and reliable software has boosted some large-scale programming to the commercial stage, computer science (which is younger than many of its researchers) has yet to build the experimental foundation on which software engineering must rest.

PROCEEDS OF PROCESS

In 1991, for example, the Software Engineering Institute, a software think tank funded by the military, unveiled its Capability Maturity Model (CMM). *"It provides a vision of software engineering and management excellence,"* beams David Zubrow, who leads a project on empirical methods at the Institute. The CMM has at last persuaded many programmers to concentrate on measuring the process by which they produce software, a prerequisite for any industrial engineering discipline.

FOREWORD *"Software's Chronic Crisis"*

Using interviews, questionnaires, and the CMM as a benchmark, evaluators can grade the ability of a programming team to create predictably software that meets its customers' needs. The CMM uses a five-level scale, ranging from chaos at Level 1 to the paragon of good management at Level 5. To date, 261 organizations have been rated.

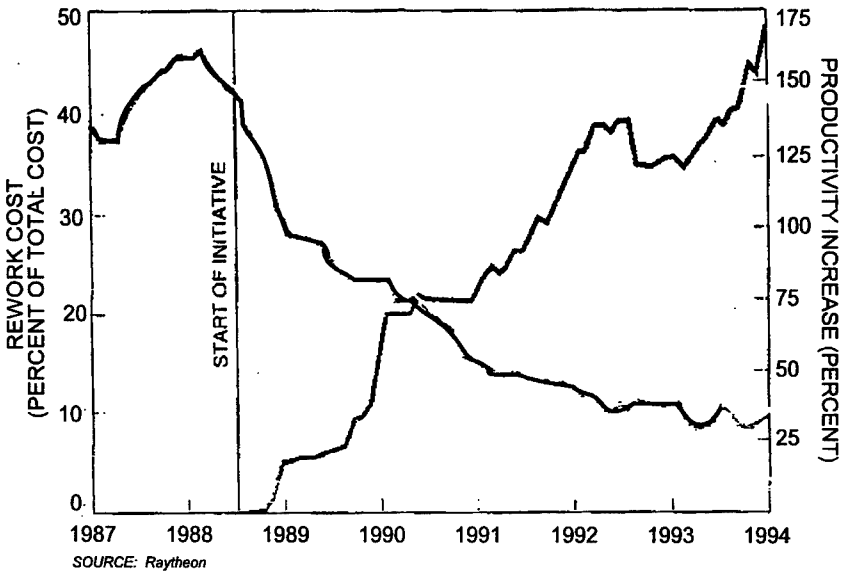
"The vast majority — about 75% — are still stuck in Level 1," Curtis reports. *"They have no formal process, no measurements of what they do, and no way of knowing when they are on the wrong track or off the track altogether."* (The Center of Naval Analysis concluded that the AAS project at IBM Federal Systems *"appears to be at a low 1 rating."*) The remaining 24% of projects are at Levels 2 or 3.

Only two elite groups have earned the high CMM rating, a Level 5. Motorola's Indian programming team in Bangalore holds one title. Loral's (formerly IBM's) on-board space shuttle software project claims the other. The Loral team has learned to control bugs so well that it can reliably predict how many will be found in each new version of the software. That is a remarkable feat, considering that 90% of American programmers do no even keep count of the mistakes they find, according to Capers Jones, chairman of Software Productivity Research. Of those who do, he says, few catch more than a third of the defects that are there.

Tom Peterson, head of Loral's shuttle software project, attributes its success to *"a culture that tries to fix not just the bug, but also the flaw in the testing process that allowed it to slip through."* Yet some bugs inevitably escape detection. The first launch of the space shuttle in 1981 was aborted and delayed for two days because a glitch prevented the five on-board computers from synchronizing properly. Another flaw, this one in the shuttle's rendezvous program, jeopardized the *Intelsat-6* satellite rescue mission in 1992.

Although the CMM is no panacea, its promotion by the Software Engineering Institute has persuaded a number of leading software companies that quantitative quality control can pay off in the long run. Raytheon's equipment division, for example, formed a *"software engineering initiative"* in 1988 after flunking the CMM test. The division began pouring $1 million per year into refining rigorous inspection and testing guidelines and training its 400 programmers to follow them.

Within three years the division had jumped two levels. By this past June, most projects — including complex radar and air traffic control systems — were finishing ahead of schedule and under budget. Productivity has more than doubled. An analysis of avoided rework cost revealed a savings of $7.80 for every dollar invested in the initiative. Impressed by such successes, the US Air Force has mandated that all its software developers must reach Level 3 of the CMM by 1998. NASA is reportedly considering a similar policy.



Raytheon has saved $17.2 million in software costs since 1988, when its equipment division began using rigorous development processes that doubled its programmers' productivity and helped them to avoid making expensive mistakes.

MATHEMATICAL RE-CREATIONS

Even the best-laid designs can go awry, and errors will creep in so long as humans create programs. Bugs squashed early rarely threaten project's deadline and budget, however. Devastating mistakes are nearly always those in the initial design that slip undetected into the final product.

Mass-market software producers, because they have no single customer to please, can take a belated and brute-force approach to bug removal: they release the faulty product as a *"beta"* version and let hordes of users dig up the glitches. According to Charles Simonyi, a chief architect at Microsoft, the new version of the Windows operating system will be beta-tested by 20,000 volunteers. That is remarkably effective, but also expensive, inefficient and — since mass-produced PC products make up less than 10% of the $92.8 billion software market in the US — usually impractical.

Researchers are thus formulating several strategies to attack bugs early or to avoid introducing them at all. One idea is to recognize that the problem a system is supposed to solve always changes as the system is being built. Denver's airport planner saddled BAE with $20 million worth of changes to the design of its baggage system long after construction had begun. IBM has been similarly bedeviled by the indecision of FAA managers. Both

companies naively assumed that once their design was approved, they would be left in peace to build it.

Some developers are at last shedding that illusion and rethinking software as something to be grown rather than built. As a first step, programmers are increasingly stitching together quick prototypes out of standard graphic interface components. Like an architect's scale model, a system prototype can help clear up misunderstandings between customer and developer before a logical foundation is poured.

Because they mimic only the outward behavior of systems, prototypes are of little help in spotting logical inconsistencies in a system's design. *"The vast majority of errors in large scale software are errors of omission,"* notes Laszlo A. Belady, director of Mitsubishi Electric Research Laboratory. And models do not make it any easier to detect bugs once a design is committed to code.

When it absolutely, positively has to be right, says Martyn Thomas, chairman of Praxis, a British software company, engineers rely on mathematical analysis to predict how their designs will behave in the real world. Unfortunately, the mathematics that describes physical systems does not apply within the synthetic binary universe of a computer program: discrete mathematics, a far less mature field, governs here. But using the still limited tools of set theory and predicate calculus, computer scientists have contrived ways to translate specifications and programs into the language of mathematics, where they can be analyzed with theoretical tools called formal methods.

Praxis recently used formal methods on an air-traffic control project for Britain's Civil Aviation Authority. Although Praxis's program was much smaller than the FAA's, the two shared a similar design problem: the need to keep redundant systems synchronized so that if one fails, another can instantly take over. *"The difficult part was guaranteeing that messages are delivered in the proper order over twin networks,"* recalls Anthony Hall, a principle consultant to Praxis. *"So here we tried to carry out proofs of our design, and they failed, because the design was wrong. The benefit of finding errors at that early stage is enormous,"* he adds. The system was finished on time and put into operation last October.

Praxis used formal notations on only the most critical parts of its software, but other software firms have employed mathematical rigor throughout the entire development of a system. GEC Alsthom in Paris is using a formal method called *"B"* as it spends $350 million to upgrade the switching-and-speed-control software that guides the 6,000 electric trains in France's national railway system. By increasing the speed of the trains and reducing the distance between them, the system can save the railway company billions of dollars that might otherwise need to be spent on new lines.

Safety was an obvious concern. So GEC developers wrote the entire design and final program in formal notation and then used mathematics to prove them consistent. *"Functional tests are still necessary, however, for*

two reasons," says Fernando Mejia, manager of the formal development section at GEC. First, programmers do occasionally make mistakes in proofs. Secondly, formal methods can guarantee only that software meets its specification, not that it can handle the surprises of the real world.

All of France's 6,000 trains will use speed- and switching-control software developed by GEC Alsthom using mathematical methods to prove that the programs are written correctly.

Formal methods have other problems as well. Ted Ralston, director of strategic planning for Odyssey Research Associates in Ithaca, N.Y., points out that reading pages of algebraic formulas is even more stultifying than reviewing computer code. Odyssey is just one of several companies that are trying to automate formal methods to make them less onerous to programmers. GEC is collaborating with Digilog in France to commercialize programming tools for the B method. The beta version is being tested by seven companies and institutions, including Aerospatiale, as well as France's atomic energy authority and its defense department.

On the other side of the Atlantic, formal methods by themselves have yet to catch on. *"I am skeptical that Americans are sufficiently disciplined to apply formal methods in any broad fashion,"* says David A. Fisher of the National Institute of Standards and Technology (NIST). There are exceptions, however, most notably among the growing circle of companies experimenting with *the "Cleanroom approach"* to programming.

The Cleanroom process attempts to meld formal notations, correctness proofs, and statistical quality control with an evolutionary approach to software development. Like the microchip manufacturing technique from which it takes its name, Cleanroom development tries to use rigorous engineering techniques to consistently fabricate products that run perfectly the first time. Programmers grow systems one function at a time and certify the quality of each unit before integrating it into the architecture.

Growing software requires a whole new approach to testing. Traditionally, developers test a program by running it the way they intend it to be used, which often bears scant resemblance to real-world conditions. In a clean-room process, programmers try to assign a probability to every execution path — correct and incorrect — that users can take. They then derive test cases from those statistical data, so that the most common paths are tested more thoroughly. Next the program runs through each test case and times how long it takes to fail. Those times are then fed back, in true engineering fashion, to a model that calculates how reliable the program is.

Early adopters report encouraging results. Ericsson Telecom, the European telecommunications giant, used Cleanroom processes on a 70 programmer project to fabricate an operating system from its telephone-

switching computers. Errors were reportedly reduced to just one per 1,000 lines of program code; the industry average is about 25 times higher. Perhaps more important, the company found that development productivity increased by 70%, and testing productivity doubled.

NO SILVER BULLET

Then again, the industry has heard tell many times before of *"Silver Bullets"* that slay werewolf projects. Since the 1960s developers have peddled dozens of technological innovations intended to boost productivity — many have even presented demonstration projects to *"prove"* the verity of their boasts. Advocates of object-oriented analysis and programming, a buzzword du jour, claim their approach represents a paradigm shift that will deliver *"a 14-to-1 improvement in productivity,"* along with higher quality and easier maintenance, all at reduced cost.

There are reasons to be skeptical. *"In the 1970s structured programming was also touted as a paradigm shift,"* Curtis recalls. *"So was CASE* [computers-assisted software engineering]. *So were third-, fourth- and fifth-generation languages. We've heard great promises for technology, many of which weren't delivered."*

Meanwhile productivity in software development has lagged behind that of more mature discipline, most notably computer hardware engineering. *"I think of software as a cargo cult,"* Cox says. *"Our main accomplishments were imported from this foreign culture of hardware engineering — faster machines and more memory."* Fisher tends to agree: adjusted for inflation, *"the value added per worker in the industry has been at $40,000 for two decades,"* he asserts. *"We're not seeing any increases."*

As CEO of Incremental Systems, David A Fisher learned firsthand why software components do not sell. Now he supervises a $150-million federal program to create a market for software parts.

"I don't believe that," replies Richard A. DeMillo, a professor at Purdue University and head of the Software Engineering Research Consortium. *"There has been improvement, but everyone uses different definitions of productivity."* A recent study published by Capers Jones — but based on necessarily dubiously historical data — states that US programmers churn out twice as much code today as they did in 1970.

The fact of the matter is that no one really knows how productive software developers are, for three reasons. First, less than 10% of American companies consistently measure the productivity of their programmers.

Second, the industry has yet to settle on a useful standard unit of measurement. Most reports, including those published in peer-reviewed computer science journals, express productivity in terms of lines-of-code per worker per month. But programs are written in a wide variety of languages and vary enormously in the complexity of their operation. Comparing the number of lines written by a Japanese programmer using C with the number produced by an American using Ada is thus like comparing their salaries without converting from yen to dollars.

Third, Fisher says, *"you can walk into a typical company and find two guys sharing a office, getting the same salary, and having essentially the same credentials and yet to find a factor of 100 difference in the number of instructions per day that they produce."* Such enormous individual differences tend to swamp the much smaller effects of technology or process improvements.

After 25 years of disappointment with apparent innovations that turned out to be irreproducible or unscalable, many researchers concede that computer science needs an experimental branch to separate the general results from the accidental. *"There has always been this assumption that if I give you a method, it is right just because I told you so,"* complains Victor R. Basili, a professor at the University of Maryland. *"People are developing all kinds of things, and it's really quite frightening how bad some of them are,"* he says.

Mary Shaw of Carnegie-Mellon points out that mature engineering fields codify proved solutions in handbooks so that even novices can consistently handle routine designs, freeing more talented practitioners for advanced projects. No such handbook yet exists for software, so mistakes are repeated on project after project, year after year.

DeMillo suggests that the government should take a more active role. *"The National Science Foundation should be interested in funding research aimed at verifying experimental results that have been claimed by other people,"* he says. *"Currently, if it's not groundbreaking, first-time-ever-done research, program officers at the NSF tend to discount the work."* DeMillo knows whereof he speaks. From 1989 to 1991 he directed the NSF's computer and computation research division.

Yet *"if software engineering is to be an experimental science. That means it needs laboratory science. Where the heck are the laboratories?"* Basili asks. Because attempts to scale promising technologies to industrial proportions so often fail, small laboratories are of limited utility. *"We need to have places where we can gather data and try things out,"* DeMillo says. *"The only way to do that is have a real software development organization as a partner."*

There have been only a few such partnerships. Perhaps the most successful is the Software Engineering Laboratory, a consortium of NASA's Goddard Space Flight Center, Computer Sciences Corporation, and the

University of Maryland. Basili helped to found the Laboratory in 1976. Since then, graduate students and NASA programmers have collaborated on *"well over 100 projects,"* Basili says, most having to do with building ground-support software for satellites.

Experimentalist Victor R. Basili helped found the Software Engineering Laboratory to push programming onto a firmer foundation of mathematics and science.

A Developing World

Since the invention of computers, Americans have dominated the software market. Microsoft alone produces more computer code each year than do any of 100 nations, according to Capers Jones of Software Productivity Research in Burlington, Mass. US suppliers hold about 70% of the worldwide software market.

But as international networks sprout and large corporations deflate, India, Hungary, Russia, the Philippines, and other poorer nations are discovering in software a lucrative industry that requires the one resource in which they are rich: an underemployed, well-educated labor force. American and European giants are now competing with upstart Asian development companies for contracts, and in response many are forming subsidiaries overseas. Indeed, some managers in the trade predict that software development will gradually split between Western software engineers who design systems and Eastern programmers who build them.

"In fact, it is going on already," says Laszlo A. Belady, director of Mitsubishi Electric Research Laboratory. AT&T, Hewlett-Packard, IBM, British Telecom and Texas Instruments have all set up programming teams in India. The Pact Group in Lyons, France, reportedly maintains a *"software factory"* in Manila. *"Cadence, the US supplier of VLSI design tools has had its software development sited on the Pacific rim for several years,"* reports Martyn Thomas, chairman of Praxis. *"ACT, a U.K.-based systems house, is using Russian programmers from the former Soviet space program,"* he adds.

So far India's star has risen fastest. *"Offshore development* [work commissioned in India by foreign companies] *has begun to take off in the past 18 to 24 months,"* says Rajendra S. Pawar, head of New Delhi-based NIIT, which has graduated 200,000 Indians from its programming courses. Indeed, India's software exports have seen a compound annual growth of 38% over the past five years; last year they jumped 60% — four times the average growth rate worldwide.

About 58% of the $360-million worth of software that flowed out of India last year ended up in the US. That tiny drop hardly makes a splash in a $92.8-billion market. But several trends may propel exports beyond the $1-billion mark as early as 1997.

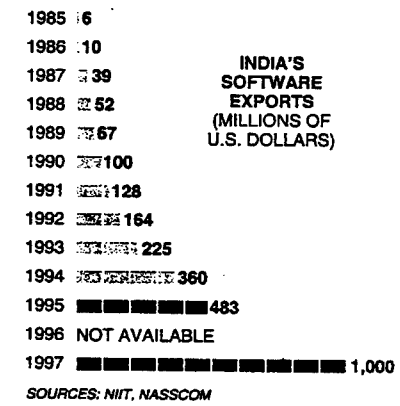
The single most important factor, Pawar asserts, is the support of the Indian government, which has eased tariffs and restrictions, subsidized numerous software technology parks and export zones, and doled out 5-year tax exemptions to software exporters. *"The opening of the Indian economy is acting as a very big catalyst,"* Pawar says.

It certainly seems to have attracted the attention of large multinational firms eager to reduce both the cost of the software they need and the amount they build in-house. The primary cost of software is labor. Indian programmers come so cheap — $125 per unit of software versus $925 for an American developer, according to Jones — that some companies fly an entire team to the US to work on a project. More than half of India's software exports come from such *"body shopping,"* although tightened US visa restrictions are stanching this flow.

Another factor, Pawar observes, is a growing trust in the quality of overseas project management. *"In the past two years, American companies have become far more comfortable with the offshore concept,"* he says. This is a result in part of success stories from leaders like Citicorp, which develops banking systems in Bombay, and Motorola, which has a top-rated team of more than 150 programmers in Bangalore building software for its iridium satellite network.

Offshore development certainly costs less than body shopping, and not merely because of saved airfare. *"Thanks to the time differences between India and the US, Indian software developers can act the elves and the shoemaker,"* working overnight on changes requested by managers the previous day, notes Richard Heeks, who studies Asian computer industries at the University of Manchester in England.

Price is not everything. Most Eastern nations are still weak in design and management skills. *"The US still has the best system architects in the world,"* boasts Bill Curtis of the Software Engineering Institute. *"At large systems, nobody touches us."* But when it comes to just writing program code, the American hegemony may be drawing to a close.



JUST ADD WATER

Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be reused at many different scales. Programmers have for decades used libraries of subroutines to avoid rewriting the same code over and over. But these components break down when they are moved to a different programming

language, computer platform, or operating environment. *"The tragedy is that as hardware becomes obsolete, an excellent expression of a sorting algorithm written in the 1960s has to be rewritten,"* observes Simonyi of Microsoft.

Fisher sees tragedy of a different kind. *"The real price we pay is that as a specialist in any software technology you cannot capture your special capability in a product. If you can't do that, you basically can't be a specialist."* Not that some haven't tried. Before moving to NIST last year, Fisher founded and served as CEO of Incremental Systems. *"We were truly world-class in three of the component technologies that go into compilers but were not as good in the other seven or so,"* he states. *"But we found that there was no practical way of selling compiler components; we had to sell entire compilers."*

So now he is doing something about that. In April, NIST announced that it was creating an Advanced Technology Program to help engender a market for component-based software. As head of the program, Fisher will be distributing $150 million in research grants to software companies willing to attack the technical obstacle that currently make software parts impractical.

The biggest challenge is to find ways of cutting the ties that inherently bind programs to specific computers and to other programs. Researchers are investigating several promising approaches, including a common language that could be used to describe software parts, programs that reshape components to match any environment, and components that have lots of optional features a user can turn on or off.

Fisher favors the idea that components should be synthesized on the fly. Programmers would *"basically capture how to do it rather than actually doing it,"* producing a recipe that any computer could understand. *"Then when you want to assemble two components, you would take this recipe and derive compatible versions by adding additional elements to their interfaces. The whole thing would be automated,"* he explains.

Even with a $150-million incentive and market pressures forcing companies to find cheaper ways of producing software, an industrial revolution in software is not imminent. *"We expect to see only isolated examples of these technologies in five to seven years — and we may not succeed technically either,"* Fisher hedges. Even when the technology is ready, components will find few takers unless they can be made cost-effective. And the cost of software parts will depend less on the technology involved than on the kind of market that arises to produce and consume them.

Brad Cox, like Fisher, once ran a software component company and found it hard going. He believes he has figured out the problem — and its solution. Cox's firm tried to sell low-level program parts analogous to computer chips. *"What's different between software ICs* [integrated circuits] *and silicon ICs is that silicon ICs are made of atoms, so they abide by conservation of mass, and people therefore know how to buy and sell them*

robustly," he says. *"But this interchange process that is at the core of all commerce just does not work for things that can be copied in nanoseconds."* When Cox tried selling the part his programmers had created, he found that the price the market would bear was far too low for him to recover the costs of development.

The reasons were twofold. First, recasting the component by hand for each customer was time consuming; NIST hopes to clear this barrier with its Advanced Technology Program. The other factor was not so much technical as cultural: buyers want to pay for a component once and make copies for free.

"The music industry has had about a century of experience with this very problem," Cox observes. *"They used to sell tangible goods like piano rolls and sheet music, and then radio and televisions came along and knocked all that into a cocked hat."* Music companies adapted to broadcasting by setting up agencies to collect royalties every time a song is aired and to funnel the money back to the artist and producers.

Cox suggests similarly charging users each time they use a software component. *"In fact,"* he says, *"that model could work for software even more easily than for music, thanks to the infrastructure advantages that computers and communications give us. Record players don't have high speed network links in them to report usage, but our computers do."*

Or will, at least. Looking ahead to the time when nearly all computers are connected, Cox envisions distributing software of all kinds via networks that link component producers, end users, and financial institutions. *"It's analogous to a credit card operation but with tentacles that reach into PCs,"* he says. Although that may sound ominous to some, Cox argues that *"the Internet now is more like a garbage dump than a farmer's market. We need a national infrastructure that can support the distribution of everything from Grandma's cookie recipe to Apple's window managers to Addison-Wesley's electronic books."* Recognizing the enormity of the cultural shift he is proposing, Cox expects to press his cause for years to come through the Coalition for Electronic Markets, of which he is president.

The combination of industrial process control, advanced technological tools, and interchangeable parts promises to transform not only how programming is done but also who does it. Many of the experts who convened at Hedsor Park agreed with Belady that *"in the future, professional people in most fields will use programming as a tool, but they won't call themselves programmers or think of themselves as spending their time programming. They will think they are doing architecture, or traffic planning, or film making."*

That possibility begs the question of who is qualified to build important systems. Today anyone can bill herself as a software engineer. *"But when you have 100 million user-programmers, frequently they will be doing things that are life critical — building applications that fill prescriptions, for example,"* notes Barry W. Boehm, director of the Center for Software

Acknowledgments

A central theme of each edition of these Guidelines is how to turn an adverse situation into a success — how to turn a challenge into a victory. The most important ingredients in achieving these goals, the most crucial resources required to produce a quality product, are the **people** selected to do the job. The skills, experience, creative abilities (or lack thereof) of the team are key determinants of success or failure in the software world.

The development and production of this book mirrors many of the challenges discussed between these pages. Lt. General W.G. Pagonis (USA, retired) talked about Moving Mountains during the Gulf War. This document represents the "*scaling of a Mount Everest*" by a very small team of highly-skilled, professional, and dedicated people. Representing over four continuous years of effort, these Guidelines were written with limited resources, a limited staff, lots of hard work, and lots of talent!

We would like to thank **Major Joseph J. Stanko** (SAF/AQRE) for overall coordination, editorial oversight, and the technical currency and accuracy of this update (Version 2.0). We would also like to convey a very special thanks to **Susan Tinch Johnson**, who has been with this document since its inception, for research, writing, editing, graphics, publication support, and software engineering, acquisition, and management insight and substance.

Stay Current

If you find these Guidelines profitable and useful, stay current with the latest developments in software engineering ***at no cost. CrossTalk*** is the DoD Journal of Software Engineering published by the Air Force Software Technology Support Center. If you do not already receive monthly issues of *CrossTalk,* you are missing out on a wealth of information and news about what's happening in the software development arena. To order *CrossTalk*, ***free-of-charge***, contact:

Customer Service
OO-ALC/TISE
7278 Fourth Street
Hill AFB, Utah 84056-5205
(801) 777-8045
Internet: custserv@software.hill.af.mil.

About These Guidelines

GUIDELINES SCOPE: Why are they so thick?

These Guidelines cover the **gamut** of software-intensive systems acquisition and management activities, from pre-program strategic planning to post-deployment software support. They are divided into three essential areas for program success. Part I, *Introduction*, lays the groundwork. To be successful in software, you must have a basic understanding of why major software acquisitions fail and succeed, the unique DoD software acquisition environment, and software-intensive life cycles and methodologies. Part II, *Engineering*, provides the meat. World-class software must be engineered, which involves implementing a series of interrelated, interdependent concepts, disciplines, and activities. These include risk management, Ada, software development maturity, measurement and metrics, reuse, software tools, and software support. Part III, *Management*, brings it all together. To successfully manage a major software-intensive systems acquisition, you must be an informed buyer. You must understand the software development process, software quality assurance, contract management techniques, and insist on continuous process improvement throughout the system's life. To step up to this challenge, you and your program will benefit by following the comprehensive software acquisition and management guidance found between these pages.

GUIDELINES AUDIENCE: Who should read them?

These Guidelines should be read by **all levels of managers** involved in the development and acquisition of major DoD software-intensive systems. Program Executive Officers (PEOs), Designated Acquisition Commanders (DACs), program managers, software maintainers and developers, engineers, action officers, technical, government, contractor, and academic personnel will all benefit from the guidance provided herein.

THESE ARE YOUR GUIDELINES: Give us your input!

For these Guidelines to remain current and meaningful they must be continuously updated and improved. Your help in citing areas for improvement and in providing practical insights and current lessons-learned from the field is vital to their usefulness and to the success of your colleagues who follow. Forward your comments and requests for additional copies of the Guidelines to:

Software Technology Support Center
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, Utah 84056-5205
(801) 777-8045
E-mail: custserv@software.hill.af.mil

TABLE OF CONTENTS

Blank page.

Table of Contents

Blank page.

VOLUME
1

Table of Contents

PREFACE
FOREWORD *"Software's Chronic Crisis"*

PART I Introduction

CHAPTER 1
Software Acquisition Overview

CHAPTER OVERVIEW 1-1
SOFTWARE VICTORY: The Exception or The Rule? 1-3
Software: The Highest Risk System Component 1-5
False Steps on the Battlefield 1-7
General Accounting Office (GAO) Reports 1-7
Scientific American Article 1-10
Battle Damage Assessments 1-11
The Parnas Papers 1-11
Fred Brooks' "*No Silver Bullet*" 1-13
1987 Defense Science Board Report on Military Software 1-15
1992 Software Process Action Team Report 1-16
1994 Defense Science Board Report on Acquiring Defense Software Commercially 1-17
WHY SOFTWARE ACQUISITIONS FAIL 1-18
Complexity 1-20
Inadequate Estimates 1-20
Size and Complexity Estimates 1-21
Cost/Schedule Estimates 1-21
Optimistic Estimates 1-22
Unstable Requirements 1-23
User Involvement 1-23

Communication 1-26
Intangibility 1-28
Complexity 1-28
Changing Threat 1-28
Poor Problem Solving/Decision-Making 1-30
Silver Bullets 1-30
Comparison of Military Software Problems with Commercial Software Problems 1-32
SOFTWARE ACQUISITION: The Bright Side 1-33
Software Success Stories 1-34
F-22 Advanced Tactical Fighter 1-35
Boeing 777 Transport 1-40
Software Best Practices 1-43
SOFTWARE ACQUISITION: Your Management Challenge 1-44
SOFTWARE ACQUISITION BOTTOM LINE 1-46
Process-Driven Acquisition Strategy 1-46
MANAGEMENT BOTTOM LINE 1-53
Leadership As Well As Management 1-53
Being a Good Leader Means Being a Good Customer 1-54
Teamwork: A New Total Force Concept 1-55
REFERENCES 1-58
ADDENDUM
SOFTWARE SOLUTION: Motorola's Strategy for Becoming The Premier Software Company 1-63

CHAPTER 2
DoD Software Acquisition Environment

CHAPTER OVERVIEW 2-1
THE INFORMATION AGE HAS DAWNED 2-3
DoD SOFTWARE DOMAINS 2-8
Weapon System Software 2-9
Embedded Software 2-10
C3 Software 2-11
Intelligence Software 2-11
Other Weapon System Software 2-12
MIS Software (Non-Weapon System Software) 2-13
ACQUISITION STREAMLINING: A National Imperative 2-14
Decreasing Budgets — Increasing Software Demands 2-15
DoD's New Order of Acquisition 2-16
Acquisition Reform Working Group 2-17
Acquisition Reform: A Mandate for Change 2-17
Federal Acquisition Streamlining Act of 1994 2-18

TABLE OF CONTENTS

MilSpec and MilStd Reform 2-19
Specifications & Standards — A New Way of Doing Business .. 2-20
MIL-STD-498, *Software Development and Documentation* 2-23
DoD Policy on the Use of Ada 2-24
C4I FOR WARRIOR 2-25
Open Systems 2-26
Technical Architecture Framework for Information Management (TAFIM) 2-27
SOFTWARE BEST PRACTICES INITIATIVE 2-32
Software Program Managers Network Products and Services .. 2-36
Training 2-36
Support Services and Products 2-38
SOFTWARE TECHNOLOGY SUPPORT CENTER (STSC) 2-41
Services 2-41
GUIDELINES, SOFTWARE BEST PRACTICES, AND YOU 2-42
REFERENCES 2-43
ADDENDUM A
MIL-STD-498: What's New and Some Real Lessons-Learned 2-47
ADDENDUM B
Adopting MIL-STD-498: The Stepping-stone to the US Commercial Standard 2-47

CHAPTER 3
System Life Cycle and Methodologies
CHAPTER OVERVIEW 3-1
LIFE CYCLE PROCESS RAISES CURTAIN ON DECISION MAKING 3-3
SYSTEM LIFE CYCLE 3-4
Life Cycle Phases, Milestone Decisions, and Activities 3-6
LIFE CYCLE MANAGEMENT METHODOLOGIES 3-11
Evolutionary 3-13
Incremental 3-16
Spiral Method 3-19
Ada Spiral Model Environment 3-20
Choosing Among Evolutionary, Incremental, and Spiral Models 3-21
Waterfall Model 3-21
Fast Track 3-22
REFERENCES 3-24

PART II Engineering

CHAPTER 4
Engineering Software-Intensive Systems

CHAPTER OVERVIEW 4-1
ENGINEERING IS THE KEY 4-3
WHAT IS DOMAIN ENGINEERING? 4-5
Domain Identification 4-7
Domain Analysis 4-7
Domain Design 4-8
Domain Implementation 4-9
Benefits of Domain Engineering 4-11
WHAT IS SYSTEMS ENGINEERING? 4-12
Integrated Product Development (IPD) 4-16
Concurrent Engineering 4-18
The Case for Software Engineering 4-20
Domain Engineering and the Software Engineering Process 4-23
Relationship Among Enterprise Engineering, Domain Engineering, and Application Engineering 4-25
WHAT IS SOFTWARE ENGINEERING? 4-26
Software Engineering Goals 4-29
Supportability 4-30
Reliability 4-30
Safety 4-31
Efficiency 4-32
Understandability 4-33
Software Engineering Principles 4-33
Abstraction and Information Hiding 4-34
Modularity and Localization 4-35
Uniformity, Completeness, and Confirmability 4-36
MANAGING SOFTWARE ENGINEERING 4-37
Software Engineering Information 4-38
WHAT IS INFORMATION ENGINEERING? 4-40
Information Engineering Process 4-42
Information Engineering Architecture 4-42
IDEF 4-44
SUCCESS THROUGH ENGINEERING 4-46
REFERENCES 4-48
ADDENDUM A
SWSC Domain Engineering Lessons-Learned 4-51
ADDENDUM B
A New Software OT&E Methodology 4-56

TABLE OF CONTENTS

CHAPTER 5
Ada: The Enabling Technology

CHAPTER OVERVIEW 5-1
Ada: BECAUSE IT'S SAFE 5-3
Ada's Eagle — Our Safest Bird of Prey 5-3
Ada: BECAUSE IT'S SMART 5-5
Ada Is for All Domains 5-5
Ada's Background 5-6
Ada: BECAUSE IT ENABLES SOFTWARE ENGINEERING 5-8
Software Engineering Principles and Ada 5-9
Ada: The Great Facilitator 5-9
Ada: FACTS AND FALLACIES 5-12
ADDENDUM A
How Long Does It Take to Learn Ada? 5-15
Ada: WHO'S USING IT? 5-19
Ada Use in DoD 5-19
Large-Scale Commercial Applications from Around the World .. 5-22
Seawolf Submarine's AN/BSY-2 Project 5-22
Demanding Software Requirements 5-26
World's Biggest, Unprecedented Ada Development 5-27
Challenging COTS Integration 5-27
Ada and Engineering Discipline Are Key to Success 5-28
BSY-2 Ada Lessons-Learned 5-29
NCTAMS-LANT Project 5-30
NCPII Re-engineering Project 5-30
Theater Display Terminal 5-30
Ada for Windows Project 5-31
Team Training Was An Innovative Effort 5-31
Lack of Ada Windows Tools Sparked More Innovation 5-32
Lessons-Learned Applied to Other Efforts 5-33
New Skills Breed New Applications and Training Programs .. 5-33
Ada Windows Team Spreads the Wealth and Presses On 5-33
Intelsat I-VII Satellite Project 5-34
Ada's Package Feature Benefits 5-35
Ada's Typing Feature Benefits 5-35
Why Ada for Intelsat I-VII? 5-36
ACHIEVABLE SUCCESS WITH ADA 5-36
Ada versus the C++ Challenge 5-37
Ada versus the Assembly Challenge 5-43
Ada versus the Fourth Generation Language (4GL) Challenge 5-45
Key Points to Consider 5-46
Ada May Be Cheaper Than 4GLs — 4GLs May Be Cheaper Than Ada 5-47

Appropriate Domains for 4GL's 5-47
Development Details 5-48
4GL Versus Ada Size Only (Impacts Of Software Technology and Environment) 5-48
What About Maintenance? 5-49
Lessons-Learned 4GL Estimation versus Ada 5-50
ADA LANGUAGE FEATURES 5-50
Ada Program Unit 5-51
Ada Subprograms 5-52
Ada's Packaging Feature 5-52
Ada's Tasking Feature 5-53
Ada's Exception Handling Feature 5-54
Ada's Generics Feature 5-55
Ada Representation Specification 5-56
Ada Input/Output Packages 5-56
Ada's Typing Feature 5-57
Ada 95: LANGUAGE FOR THE 21ST CENTURY 5-58
Ada IMPLEMENTATION 5-62
Transitioning to Ada 95 5-62
Adopt An Incremental Transition Strategy 5-63
Write Ada 95-Compatible Code in Ada 83 5-64
Remember the Human Factor 5-64
Ada TECHNOLOGY CONSIDERATIONS 5-65
Ada Compilers 5-65
Compiler Selection 5-65
Compiler Maturity 5-66
Compiler Validation 5-66
Compiler Evaluation 5-67
Compiler Benchmarks 5-67
Choosing the Appropriate Ada 95 Compiler 5-68
Ada Interface Standards 5-69
Ada Bindings 5-70
Operating Systems 5-72
Databases 5-72
Graphics 5-73
Windowing Environment 5-73
Ada Run-time Efficiency 5-73
Ada AND YOUR PROGRAM 5-74
Ada and Your New-Start Program 5-74
Ada and Your On-going Program 5-75
Ada and Your PDSS Program 5-75
Ada Upgrade Opportunities 5-76
Mixing Ada with Other Languages 5-76
Porting with Ada 5-76

TABLE OF CONTENTS

Requirements and Design Impacts 5-77
ADDRESSING Ada IN THE RFP 5-78
Ada Waivers 5-79
Ada: THE LANGUAGE OF CHOICE 5-81
REFERENCES 5-83
ADDENDUM B
Ada Users Throughout the World Tell: WHY Ada! 5-87
ADDENDUM C
The Ada 95 Philosophy 5-93
ADDENDUM D
Ada Implementation Lessons-Learned from SSC and CSC ... 5-93

CHAPTER 6
Risk Management
CHAPTER OVERVIEW 6-1
RISK MANAGEMENT: An Investment in Success 6-3
SOFTWARE RISK 6-5
Software Risk Factors 6-6
Common Risk Factors 6-6
"Top-10" Risk Identification Checklist 6-8
Environmental Factors 6-8
Interrelated Factors 6-9
MANAGING SOFTWARE RISK 6-9
Risk Mitigation Techniques 6-10
Risk Management Process 6-11
FORMAL RISK MANAGEMENT METHODS 6-12
Software Risk Evaluation (SRE) Method 6-13
SRE Functional Components 6-14
Harris Corporation SRE Risk Management Streamlining Example 6-18
Boehm's Software Risk Management Method 6-19
Risk Management Paradigm 6-20
Boehm's Risk Management Process 6-20
Best Practices Initiative's Risk Management Method 6-24
Team Risk Management Method 6-27
Team Risk Management Principles 6-27
Team Risk Management Advantages 6-27
Team Risk Management on the NALCOMIS Program Example 6-29
B-1B Computer Upgrade Risk Management Method and Example 6-31
Identified Risks 6-31

Contractor Risk Management Teams 6-33
Program Office Estimate 6-33
Integrated Risk Management Process 6-34
Chief Engineers' Watchlist 6-34
Computer Resources Working Group 6-35
RISK MANAGEMENT PLANNING 6-35
Risk Management Plan 6-36
Contingency Planning 6-38
Crisis Management Plan 6-40
Crisis Recovery Plan 6-41
RISK ELEMENT TRACKING 6-41
Risk Tracking Methods 6-42
ADDRESSING RISK IN THE RFP 6-43
Offeror's Risk Methodology 6-43
Risk-Based Source Selection 6-44
Performance Risk Analysis Group (PRAG) 6-46
SOFTWARE RISK MANAGEMENT BEGINS WITH YOU! 6-47
REFERENCES 6-47

CHAPTER 7
Software Development Maturity

CHAPTER OVERVIEW 7-1
PROCESS MATURITY: An Essential for Success 7-3
SOFTWARE DEVELOPMENT CAPABILITY ASSESSMENT METHODS 7-4
Software Development Capability Evaluation (SDCE) 7-7
SEI Software Capability Evaluation (SCE) 7-7
MATURITY MODELS 7-9
Capability Maturity Model (CMM[SM]) 7-11
ISO/IEC Maturity Standard: SPICE 7-13
SPICE Product Suite 7-14
Baseline Practices Guide 7-14
BPG Capability Levels 7-15
Common Features and Generic Practices 7-16
People — Capability Maturity Model (P-CMM[SM]) 7-18
P-CMM[SM] Structure 7-19
Software Acquisition — Capability Maturity Model (SA-CMM[SM]) 7-21
Systems Security Engineering — Capability Maturity Model (SSE-CMM[SM]) 7-23
Trusted Software — Capability Maturity Model (TS-CMM[SM]) 7-26
Systems Engineering — Capability Maturity Model (SE-CMM[SM]) 7-26

TABLE OF CONTENTS

SE-CMM[SM] Architecture 7-27
BENEFITS OF MOVING UP THE MATURITY SCALE 7-29
Moving Up the Maturity Scale at USAISSDCL 7-32
Moving Up the Maturity Scale at OC-ALC 7-34
Moving Up the Maturity Scale at USSTRATCOM 7-35
Evidence of Improvement 7-36
Methods for Success 7-37
Process Maturity Pays Off 7-38
Moving Up the Maturity Scale at SSG 7-38
Moving Up the Maturity Scale at Raytheon's Equipment Division, Software Systems Lab 7-40
Moving Up the Maturity Scale at Hughes Aircraft Company, Software Engineering Division 7-42
Moving Up the Maturity Scale at Litton Data Systems 7-44
ADDRESSING MATURITY IN THE RFP 7-46
REFERENCES 7-47
ADDENDUM A
A Correlation Study of the CMM[SM] and Software Development Performance 7-49
ADDENDUM B
Lessons-Learned While Achieving a CMM[SM] Level 3 Rating .. 7-63

CHAPTER 8
Measurement and Metrics

CHAPTER OVERVIEW 8-1
MEASUREMENT 8-3
Measures, Metrics, and Indicators 8-3
Software Measurement Life Cycle 8-7
Software Measurement Life Cycle at Loral 8-9
SOFTWARE MEASUREMENT PROCESS 8-11
Metrics Usage Plan 8-12
Boeing 777 Metrics Program 8-14
Metrics Selection 8-17
Data Collection 8-18
Data Analysis 8-20
National Software Data and Information Repository (NSDIR) ... 8-22
Why the NSDIR Is Necessary 8-24
NSDIR History 8-25
TYPICAL SOFTWARE MEASUREMENTS AND METRICS 8-27
Quality 8-28
User Satisfaction 8-29
Size 8-32

Measuring Software Size 8-33
Source Lines-of-Code Estimates 8-34
Function Point Size Estimates 8-35
Feature Point Size Estimates 8-37
Complexity 8-38
Requirements 8-40
Effort 8-41
Productivity 8-42
Cost and Schedule 8-44
Cost and Schedule Estimation Methodologies/Techniques 8-46
Ada-Specific Cost Estimation 8-49
Scrap and Rework 8-49
Support 8-50
CAUTIONS ABOUT METRICS 8-51
ADDRESSING MEASUREMENT IN THE RFP 8-53
REFERENCES 8-54
ADDENDUM A
Assessment Metrics for Use with the Capability Maturity Model: Are We Improving? 8-57
ADDENDUM B
Software Complexity 8-63
ADDENDUM C
Metrics: The Measure of Success 8-63
ADDENDUM D
Making Metrics Work Miracles 8-64
ADDENDUM E
Swords and Plowshares: The Rework Cycles of Defense & Commercial Software Development Projects 8-64

CHAPTER 9
Reuse

CHAPTER OVERVIEW 9-1
INCREASED QUALITY THROUGH REUSE 9-3
REUSE PROCESS 9-4
Reuse Management Systems (Repositories) 9-7
IMPLEMENTING REUSE 9-7
Product-Line Approach 9-9
Product-Line Benefits 9-10
Product-Line Paradigm Shift 9-11
OPPORTUNITIES FOR REUSE 9-12
Reuse for Embedded Weapon Systems 9-12
Specification Reuse 9-15

TABLE OF CONTENTS

Architecture Reuse 9-15
Design Reuse 9-16
Code Reuse 9-16
Ada Reuse 9-18
Data Reuse 9-18
COST/BENEFITS OF REUSE 9-18
Cost of Reuse 9-19
Benefits of Reuse 9-20
Reuse at the Standard Systems Center (SSC) 9-22
Reuse on the F-22 Program 9-23
Reuse on the F-16 Upgrade Program 9-24
Reuse at Hewlett-Packard (HP) 9-24
REUSE PROGRAMS 9-26
Software Technology for Adaptable, Reliable Systems (STARS) 9-26
STARS Space Command and Control Architecture Infrastructure (SCAI) 9-29
Asset Source for Software Engineering Technology (ASSET) .. 9-34
Comprehensive Approach to Reusable Defense Software (CARDS) 9-35
Portable Reusable Integrated Software Modules (PRISM) 9-38
Defense Software Repository System (DSRS) 9-38
Electronic Library Services and Applications (ELSA) 9-39
Interoperability Among Software Reuse Libraries 9-39
DSSA ADAGE Program 9-39
RICC 9-41
ADDRESSING REUSE IN THE RFP 9-42
A FINAL WORD ON REUSE 9-43
REFERENCES 9-44

CHAPTER 10
Software Tools

CHAPTER OVERVIEW 10-1
PRODUCTION EFFICIENCY THROUGH TOOLS, METHODS, AND MODELS 10-3
Tool Process Improvement Benefits 10-6
MANAGING NEW TECHNOLOGIES 10-8
Technology Strategy 10-11
Technology Selection 10-13
Typical Toolset 10-14
More Cautions About CASE Tools 10-18
Technology Transition 10-19

TABLE OF CONTENTS

SOFTWARE ENGINEERING TOOLS AND ENVIRONMENTS .. 10-20
CASE Tools 10-21
Software Engineering Environments (SEEs) 10-21
UNAS 10-22
CCPDS-R Ada Success Story/UNAS Tool Design 10-24
Rational Apex™ 10-27
ASC/SEE 10-28
COHESION™ Team/SEE 10-28
Demonstration Project SEE 10-29
I-CASE 10-32
Ada-ASSURED 10-33
PROGRAM MANAGEMENT TOOLS, METHODS, AND MODELS .....
..... 10-35
Cost/Schedule/Size Estimation Models 10-37
Parametric Model Selection and Use 10-37
Cost Analysis Requirements Document (CARD) 10-40
Air Force Acquisition Model (for Weapon System Software) ... 10-40
Program Management Support System (PMSS) 10-42
ADARTS® 10-45
Process Weaver® 10-46
REQUIREMENTS AND DESIGN TOOLS 10-47
Requirements Engineering Environment (REE) 10-47
Sammi Development Kit (SDK) 10-49
AdaSAGE 10-49
AdaSAGE Success Story 10-51
Teamwork®/Ada 10-52
Common Object Request Broker Architecture (CORBA) 10-53
TESTING TOOLS 10-55
Rate Monotonic Analysis for Real-Time Systems 10-55
AdaQuest 10-58
AdaWISE 10-59
AdaTEST 10-60
MathPack 10-60
McCabe Design Complexity Tool 10-60
McCabe Instrumentation Tool 10-62
McCabe Slice Tool 10-62
Analysis of Complexity Tool (ACT) 10-62
Battlemap Analysis Tool (BAT) 10-62
TestMate 10-63
RE-ENGINEERING TOOLS 10-64
SORTS 10-65
MEASUREMENT TOOLS 10-66
AdaMAT 10-66
Amadeus Measurement System 10-67

TABLE OF CONTENTS

CONFIGURATION MANAGEMENT TOOLS 10-69
Process Configuration Management Software (PCMS) 10-70
TECHNOLOGY SUPPORT PROGRAMS 10-71
Ada Joint Program Office 10-71
Ada Information Clearinghouse (AdaIC) 10-72
PAL 10-72
ASIS 10-73
Advanced Computer Technology (ACT) Program 10-74
Computer Resource Technology Transition (CRTT) Program 10-74
Embedded Computer Resources Support Improvement Program ... (ESIP) 10-74
TECHNOLOGY SUPPORT CENTERS 10-75
Software Technology Support Center (STSC) 10-75
Standard Systems Center (SSC) 10-76
Software Engineering Institute (SEI) 10-76
Rome Laboratory 10-77
Oregon Graduate Institute Formal Methods Research 10-78
ADDRESSING TOOLS IN THE RFP 10-79
REFERENCES 10-80
ADDENDUM A
COTS Integration and Support Model 10-83
ADDENDUM B
Rate Monotonic Analysis: Did You Fake It? 10-99

CHAPTER 11
Software Support

CHAPTER OVERVIEW 11-3
SOFTWARE SUPPORT: A Total Life Cycle Approach 11-3
Software Support Cost Drivers 11-5
Software Support Activities 11-6
Software Support Issues 11-7
COTS Software Support Issues 11-10
PLANNING FOR SUPPORT SUCCESS 11-11
Software Support Cost Estimation 11-13
SOFTWARE RE-ENGINEERING 11-13
Re-engineering Decision 11-14
Re-engineering Process 11-15
Re-engineering to Ada 11-17
Re-engineering COBOL at WPFA 11-18
STSC Re-engineering Support 11-19
LOGISTICS SUPPORT ANALYSIS (LSA) 11-19
LSA on the F-22 Program 11-21

CONTINUOUS ACQUISITION AND LIFE CYCLE SUPPORT (CALS) 11-24
MANAGING A PDSS PROGRAM 11-25
Computer Resources Integrated Support Document (CRISD) 11-27
Society of Automotive Engineers (SAE) 11-28
ADDRESSING SOFTWARE SUPPORT IN THE RFP 11-29
Specifying Supportable Software 11-31
Statement of Objectives (SOO) 11-31
Specification Practices 11-31
Documentation 11-32
Life Cycle Software Support Strategies 11-33
REFERENCES 11-34
ADDENDUM A
ROADS: The "*Software Logistics Vehicle*" for the Digitized Battlefield 11-37
ADDENDUM B
Electronic Combat Model Re-engineering 11-44

PART III Management

CHAPTER 12
Strategic Planning: The Ounce of Prevention

CHAPTER OVERVIEW 12-1
PLANNING IS KEY TO SUCCESS 12-3
STRATEGIC PLANNING GOALS 12-6
Program Stability 12-8
Quality 12-9
On-Time Completion/Within Budget 12-10
SOFTWARE ACQUISITION STRATEGY 12-12
Mission Definition 12-14
Acquisition Strategy Development 12-15
Competition 12-16
Concurrency/Time Phasing 12-17
Design-to-Cost 12-18
Performance Demonstrations 12-19
Performance Incentives 12-20
Make-or-Buy 12-21
Pre-planned Product Improvement (P3I) 12-22
PROGRAM PLANNING PROCESS 12-23
Planning Objectives 12-26
Planning Scope 12-26

TABLE OF CONTENTS

Recommendations for Program Planners 12-27
PROGRAM DECOMPOSITION 12-28
System/Segment Specification (SSS) 12-29
Work Breakdown Structure (WBS) 12-29
WBS Interrelationships 12-29
Prime Mission Product Summary WBS 12-31
Software Project Summary WBS 12-31
Software Contract WBS 12-33
Software Project WBS 12-35
MARKET ANALYSIS 12-36
Software Product Definition and Decomposition 12-37
BASELINE ESTIMATES 12-37
Estimation Accuracy 12-40
Program Estimate Selection 12-42
CONTINUOUS PROGRAM PLANNING 12-42
Continuous Planning Recommendations 12-44
OTHER PLANNING CONSIDERATIONS 12-45
Major Milestones and Baselines 12-45
Program Budgeting and Funding 12-49
REFERENCES 12-50

CHAPTER 13
Contracting for Success
CHAPTER OVERVIEW 13-1
TEAM BUILDING: ATTACKING THE LION 13-3
Building High-Performance Teams 13-5
J-CALS Teamwork in Action 13-7
CONTRACT TYPE SELECTION 13-8
DEVELOPING THE RFP 13-10
RFP Development Team Building 13-11
Statement of Work (SOW) 13-13
Contractual Data Requirements List 13-14
Subcontracting 13-14
Joint Venture Partnerships 13-17
SPECIAL SOFTWARE RFP CONSIDERATIONS 13-18
Commercial-off-the-Shelf (COTS) Software 13-21
COTS Advantage 13-21
COTS Integration 13-23
COTS Integration with Ada 13-26
COTS Integration Lessons-Learned 13-27
More Cautions About COTS 13-29
Cautions About Modifying COTS 13-31

Data Rights 13-34
SPECIAL SOFTWARE SOURCE SELECTION CRITERIA 13-36
Key Software Development Personnel 13-37
Skills Matrix 13-38
Management Commitment 13-38
SOURCE SELECTION 13-39
Source Selection Team Building 13-40
Source Selection Planning 13-41
Pre-Validation Phase 13-41
Selecting the Last Team Member 13-42
Navy Seawolf Lessons-Learned 13-44
Proposal Evaluation 13-45
Best-Value versus the Cost of Poor Quality 13-46
PROTESTS 13-48
CONTRACT AWARD 13-51
REFERENCES 13-53
ADDENDUM A
Lessons-Learned in the GSA Trailboss Course 13-57
ADDENDUM B
Contracting for Success 13-80

CHAPTER 14
Managing Software Development

CHAPTER OVERVIEW 14-1
WINNING THE BATTLE WITH QUALITY 14-3
SOFTWARE DEVELOPMENT PROCESS 14-4
Systems Perspective 14-5
Design Management and Review 14-6
Effective Teamwork with Clear Roles 14-7
Process-Approach to Quality 14-7
Software Development Plan (SDP) 14-8
Software Development Recommendations 14-9
Lessons-Learned from SSC and CSC 14-11
SOFTWARE REQUIREMENTS 14-15
Software Requirements Specification (SRS) 14-18
Interface Requirements Specification (IRS) 14-19
Requirements Management 14-20
Prototyping 14-21
Prototyping Benefits 14-23
Cautions About Prototypes 14-23
HARDWARE REQUIREMENTS 14-25
VHSIC/ VHDL 14-26

TABLE OF CONTENTS

Hardware Selection 14-26
DESIGN 14-28
Design Simplicity 14-30
Architectural Design 14-31
Addressing Architecture in the RFP 14-35
Preliminary Design Review (PDR) 14-36
Detailed Design 14-38
Functional Design 14-39
Data-Oriented Design 14-39
Object-Oriented Design 14-40
Object-Oriented Baseball 14-41
Problem Domains and Solution Domains 14-45
Critical Design Review (CDR) 14-47
TESTING 14-50
Testing Objectives 14-51
Defect Detection and Removal 14-52
Defect Removal Strategies 14-55
Developer Testing 14-55
Unit Testing 14-57
Cautions About Unit Testing 14-59
Integration Testing 14-59
System Testing 14-60
Government Testing 14-60
AFOTEC Testing Objectives 14-62
Usability 14-63
Effectiveness 14-63
Software Maturity 14-64
AFOTEC Software Evaluation Tools 14-64
AFOTEC Lessons-Learned 14-65
IMPLEMENTATION 14-67
BUILDING SECURE SOFTWARE 14-68
Security Planning 14-68
Operations Security (OPSEC) 14-70
SOFTWARE DOCUMENTATION 14-75
Must-Have Documentation 14-77
REFERENCES 14-80
ADDENDUM A
The Multilevel Information Systems Security Initiative 14-83
ADDENDUM B
If Architects Had to Work Like Programmers 14-89
ADDENDUM C
On Board Software for the Boeing 777 14-92

CHAPTER 15
Managing Process Improvement

CHAPTER OVERVIEW 15-1
ATTAINING THE QUALITY OBJECTIVE 15-3
PROGRAM/CONTRACT MANAGEMENT 15-11
Cost/Schedule Control System Criteria (C/SCSC) 15-15
Earned-Value Software Metrics 15-21
Lessons-Learned from the Navy Seawolf Program 15-26
Lessons-Learned from SSC and CSC 15-26
SOFTWARE QUALITY ASSURANCE 15-28
Defect Prevention 15-32
Defect Causal Analysis 15-33
Defect Removal Efficiency 15-37
REVIEWS, AUDITS, AND INSPECTIONS 15-38
Peer Inspections 15-39
Cost and Quality Benefits of Inspections 15-41
Formal Peer Inspection Process 15-44
Peer Inspection Case Study 15-46
Peer Inspection Buy-In 15-47
Peer Inspection Training 15-48
Independent Verification & Validation (IV&V) 15-49
CLEANROOM ENGINEERING 15-49
Correctness Verification 15-51
Cleanroom Engineering Results 15-53
Cleanroom for New-Start Programs 15-55
Cleanroom for On-Going Programs 15-56
Cleanroom for Troubled Programs 15-56
Cleanroom Training 15-56
Cleanroom Information 15-56
IMPROVING PRODUCTIVITY 15-57
Ada Use 15-59
Reuse 15-60
Design Simplicity 15-60
User Involvement 15-60
Prototyping 15-61
Automated Tools 15-61
Software Productivity Consortium 15-62
TRAINING 15-63
Lessons-Learned from SSC and CSC 15-66
CONFIGURATION MANAGEMENT 15-66
Configuration Management with Ada 15-71
REFERENCES 15-72

TABLE OF CONTENTS

ADDENDUM A
Improving Software Economics in the Aerospace and Defense ... Industry 15-75
ADDENDUM B
Training — Your Competitive Edge in the '90s 15-99
ADDENDUM C
Lessons-Learned from BSY-2's Trenches 15-100

CHAPTER 16
The Challenge
CHAPTER OVERVIEW 16-1
SEIZE THE OPPORTUNITY 16-3
Embrace the Software Vision: Make It Work for You 16-4
Make the Commitment to Excellence 16-7
PROGRAM MANAGEMENT CHALLENGE 16-8
Managing a New-Start Program 16-9
Managing an On-going Program 16-9
If It Ain't Broke, Break It! 16-10
Introducing New Processes, Methods, and Tools 16-14
Managing a PDSS Program 16-15
Determining If Your Program is in Trouble 16-16
What to Do With a Troubled Program 16-18
Increase Your Schedule 16-23
Reduce Your Software Size 16-25
Improve Your Process 16-26
What To Do With a Program Catastrophe? 16-27
Abandoning the Catastrophe 16-27
THE CONTINUOUS IMPROVEMENT CHALLENGE 16-28
Measurement 16-28
Baselines 16-29
Benchmarks 16-29
Texas Instruments (TI) Benchmarking Process 16-31
YOUR MANAGEMENT CHALLENGE 16-32
REFERENCES 16-34
ADDENDUM
Reflections on Success 16-37

LIST OF FIGURES

Figure 1-1 Defect Rework Hidden Cost 1-21
Figure 1-2 Software Cost Estimation Accuracy versus Phase 1-24
Figure 1-3 Error Propagation Cost 1-27
Figure 1-4 Software Changes Convert B1-B from Nuclear to Conventional Capability 1-29
Figure 1-5 F-22 Flagship Acquisition Program 1-36
Figure 1-6 F-22 versus F-15 Life Cycle Cost Savings 1-38
Figure 1-7 Boeing 777 Transport Within Budget and On Time 1-41
Figure 2-1 F/A-18 Hornet Blasting Off Deck of USS Saratoga 2-4
Figure 2-2 Without Software the F-16 Is Only a 15-Million Dollar Lawn Dart 2-5
Figure 2-3 Software-Intensive Systems Growth 2-6
Figure 2-4 C-17 Aces EMD Phase with Simulated Durability Testing 2-8
Figure 2-5 F-16 Embedded Software Iceberg 2-11
Figure 2-6 C3I Software Provides Secure Information to Tactical Operations .. 2-12
Figure 2-7 Other Weapon System Software (Not Embedded) 2-13
Figure 2-8 TAFIM Contents 2-28
Figure 2-9 TAFIM Technical Reference Model 2-30
Figure 2-10 Technical Architecture Framework 2-31
Figure 2-11 Architectural Modeling Framework 2-31
Figure 2-12 DoD Goal Security Architecture 2-32
Figure 2-13 Human Computer Interface (HCI) Style Guide 2-33
Figure 3-1 Software-Intensive System 3-5
Figure 3-2 Nominal Cost Distribution of a Typical DoD Program 3-8
Figure 3-3 Effect of Early Decisions on Life Cycle Cost 3-8
Figure 3-4 Evolutionary Life Cycle Generations 3-14
Figure 3-5 User Involvement in the Evolutionary Method 3-15
Figure 3-6 Example Incremental Life Cycle Method 3-17
Figure 3-7 Example Incremental Method/MAISRC/Project Board Milestones/ Reviews 3-18
Figure 3-8 Ada Spiral Model 3-19
Figure 4-1 Total Quality Engineering 4-5
Figure 4-2 Vertical and Horizontal Domains 4-6
Figure 4-3 Systems Engineering Process 4-13
Figure 4-4 Relationship between Systems, Hardware, and Software Engineering 4-15
Figure 4-5 Example Integrated Product Team Members 4-16
Figure 4-6 Integrated Product Development Process 4-17
Figure 4-7 Order of Magnitude Between Software Engineering and Software-as-Art 4-21
Figure 4-8 Software Life Cycle Costs 4-23
Figure 4-9 Domain Engineering and Software Engineering Discipline 4-24
Figure 4-10 Three-tiered View of Organizational Engineering Processes 4-26
Figure 4-11 Software Engineering Elements 4-29
Figure 4-12 Software Engineering Relationship to the Software Life Cycle 4-39
Figure 4-13 Information Engineering Phases 4-42
Figure 4-14 Strategic Management Planning 4-43
Figure 4-15 Information Engineering Four-Level Architecture 4-43
Figure 4-16 IDEF (Level A0) for Nominal Program 4-45
Figure 4-17 IDEF (Decomposition of A0) Model for a Nominal Program 4-46
Figure 4-18 Software Engineering Builds a Solid Foundation Upfront 4-47
Figure 4-19 SWSC Software Re-engineering Environment 4-52
Figure 4-20 SWSC Domain Engineering Approach 4-52
Figure 4-21 Domain/Application Model Relationship 4-53

TABLE OF CONTENTS

Figure 4-22 Iterative Two Life Cycle Domain/Application Engineering Process . 4-54
Figure 5-1 Ada's Eagle: Our Safest Fighter 5-4
Figure 5-2 Ada Component Interfacing Facilitated YF-22 Fire Control Software Demonstration 5-10
Figure 5-3 Software-Intensive SSN-21 Seawolf Attack Stealth Submarine 5-26
Figure 5-4 Type Programs Where Ada Reuse is Cheaper Than 4GL 5-49
Figure 5-5 Ada Features Support Software Engineering Principles 5-51
Figure 5-6 Ada Two-part Program Unit 5-52
Figure 5-7 An Ada Package 5-53
Figure 5-8 Searching for the Exception Handler 5-55
Figure 5-9 Relationships between Ada's Features, Software Engineering Principles and Goals 5-58
Figure 5-10 Ada Bindings 5-71
Figure 5-11 Linking Ada to Non-Ada Modules 5-77
Figure 5-12 Summary of Ada Feature Benefits 5-81
Figure 6-1 Benefits of Effective Risk Management 6-5
Figure 6-2 Risk Management Continuous Process 6-11
Figure 6-3 A Formal Risk Management Process 6-12
Figure 6-4 SRE Method Application 6-14
Figure 6-5 Structure of the Software Risks Taxonomy 6-15
Figure 6-6 SRE Functional Components 6-16
Figure 6-7 Risk Magnitude Level Matrix 6-17
Figure 6-8 Streamlined SEI Risk Assessment Process 6-19
Figure 6-9 Decision Tree for Satellite Software Risk Item 6-21
Figure 6-10 Software Risk Management Steps 6-22
Figure 6-11 NALCOMIS Local Repair Cycle 6-29
Figure 6-12 One Third of All Navy/Marine Squadrons Use NACOMIS OMA 6-30
Figure 6-13 Risk Management Filters Through All B-1B Upgrade Phases 6-32
Figure 6-14 Contractor Risk Management Teams 6-33
Figure 7-1 A Family of CMM[SM]s 7-10
Figure 7-2 Software Process Maturity Framework 7-12
Figure 7-3 Three Necessary Components for Improvement 7-18
Figure 7-4 P-CMM[SM] Key Process Areas by Maturity Level 7-20
Figure 7-5 SA-CMM[SM] Architecture 7-22
Figure 7-6 System Security Engineering CMM[SM] KPAs 7-24
Figure 7-7 SE-CMM[SM] Architecture 7-28
Figure 7-8 Reduction per Year in Post-release Defect Reports 7-31
Figure 7-9 Gain per Year in Productivity 7-32
Figure 7-10 Return on Investment Ratio of SPI Efforts 7-32
Figure 7-11 Benefit Ratios of Moving from Level 1 to Level 3 7-39
Figure 7-12 1994 Assessment Results 7-44
Figure 7-13 Characteristics of a Level 3 Contractor 7-46
Figure 7-14 The Effect of Process Maturity on Performance 7-52
Figure 7-15 Origin of Data Points 7-55
Figure 7-16 Scatter Plot of CPI versus Rating for High and Very High Rating Relevance 7-57
Figure 7-17 Scatter Plot of SPIU versus Rating for Less Than 80% Complete .. 7-58
Figure 8-1 Software Measurement Life Cycle 8-7
Figure 8-2 Organizational Measurement Hierarchy 8-12
Figure 8-3 Space Shuttle Life Cycle Measurements 8-13
Figure 8-4 Total Metrics Roll-Up for the 777 ..... 8-14, 8-15
Figure 8-5 ATF Data Collection for Software Development Tracking 8-19
Figure 8-6 CMM Maturity Level, Measures Collected, Metrics Compared 8-21
Figure 8-7 Management Process with Metrics 8-22
Figure 8-8 Mission Availability Satisfaction 8-31

TABLE OF CONTENTS

Figure 8-9 Warfighting Supportability Requirements Satisfaction 8-31
Figure 8-10 Operating Cost Satisfaction (total cost/installation) 8-31
Figure 8-11 Function Point Software Size Computational Process 8-36
Figure 8-12 Coupling and Cohesion versus Fault Rate 8-40
Figure 8-13 Software Productivity Factors (Weighted Cost Multipliers) 8-43
Figure 9-1 Domain-specific, Architecture-based Software Engineering to Maximize Reuse 9-5
Figure 9-2 Reusable Asset Production through Iterative Domain and Application Engineering 9-6
Figure 9-3 Factory Fixed and Variable Costs 9-9
Figure 9-4 Design Reuse Compared to Code Reuse 9-17
Figure 9-5 Annual Projected DoD Software Cost (dollars in billions) 9-21
Figure 9-6 SEL Comparisons of Defect Rates and Development Cost of New and Reused Modules 9-22
Figure 9-7 STARS Megaprogramming 9-27
Figure 9-8 Process-based Development Conceptual Model 9-28
Figure 9-9 SWSC Stovepiped Software Systems 9-30
Figure 9-10 Architectural Infrastructure is the Product-Line Approach Foundation 9-31
Figure 9-11 SWSC Product-Line Saved People Resources 9-31
Figure 9-12 Reuse on SCAI Mobile Space System Build 2 9-32
Figure 9-13 Architecture Is the Key to Reuse Success 9-33
Figure 9-14 Obstacles to the Product-Line Paradigm Shift 9-34
Figure 9-15 High-Level Architecture View of DSSA ADAGE 9-41
Figure 10-1 Gains in Quality and Productivity Attained by Technology Transitions 10-12
Figure 10-2 Connectivity Among Software Engineering Toolboxes 10-17
Figure 10-3 CCPDS-R Time Savings Approach 10-26
Figure 10-4 COHESION SEE Functions 10-29
Figure 10-5 Demonstration Project SEE Tool Functionality Groups 10-30
Figure 10-6 I-CASE Operational Concept 10-33
Figure 10-7 Teamwork®/Ada Overview 10-54
Figure 10-8 Benefits of Automated Code Verification 10-59
Figure 10-9 Re-engineering Process Models 10-64
Figure 10-10 COTS Integration and Support 10-84
Figure 10-11 Project Model 10-87
Figure 11-1 Support Tasks Superimposed on the Software Development Phase 11-4
Figure 11-2 Life Cycle Support Costs 11-5
Figure 11-3 Causes of Software Changes 11-6
Figure 11-4 AFOTEC Software Supportability Evaluation Areas 11-7
Figure 11-5 Bathtub Curves for Hardware and Software 11-8
Figure 11-6 Relationship Among Support Engineering Tasks 11-16
Figure 11-7 Post-Deployment Software Support Key Considerations 11-25
Figure 11-8 Computer Resources Integrated Support Document (CRISD) 11-27
Figure 11-9 Acquisition Instruments 11-31
Figure 11-10 Prototype ROADS Model Configuration 11-40
Figure 11-11 ROADS Concept of Operation Constant Source System Architecture 11-41
Figure 11-12 ROADS Deployment Examples 11-42
Figure 12-1 Factors Influencing Program Stability 12-7
Figure 12-2 Interrelationships Among WBS Types 12-30
Figure 12-3 Interrelationships Among WBS Types 12-32
Figure 12-4 Software Project and Contract WBS Functional Integration 12-34
Figure 12-5 F-22 3-Level WBS 12-35
Figure 12-6 Iterative Software Planning Process 12-43

TABLE OF CONTENTS

Figure 12-7 Maturity Growth Curve 12-48
Figure 13-1 Typical Software Acquisition/Development Team 13-4
Figure 13-2 Contracting Process 13-6
Figure 13-3 RFP Preparation Process 13-11
Figure 13-4 Prime/Software Subcontractor Development Responsibility 13-15
Figure 13-5 Prime-Sub Different Business Perspectives 13-16
Figure 13-6 Chain of Government-Subcontractor Communications 13-16
Figure 13-7 Source Selection Preparation Process 13-41
Figure 13-8 Contract Award Process 13-52
Figure 14-1 F-22 Requirements Process 14-17
Figure 14-2 F-22 Baselined Requirements with Incremental Buildup 14-21
Figure 14-3 Ingredients of Software Design 14-29
Figure 14-4 Standards-Based Architecture Planning Process 14-33
Figure 14-5 Object-Oriented Baseball 14-41
Figure 14-6 Aggregation Hierarchy Example 14-42
Figure 14-7 Classification Hierarchy Example 14-44
Figure 14-8 Message Passing Example 14-44
Figure 14-9 Problem Domain/Solution Domain Analytical Process 14-45
Figure 14-10 Space Shuttle Defect Detection/Removal/Prevention Is Critical 14-46
Figure 14-11 Benefits of Using an OOD Approach with Ada 14-48
Figure 14-12 Space Shuttle Defect Removal Process Improvement 14-54
Figure 14-13 F-16 Avionics System Integration Testing 14-61
Figure 14-14 F-16 Avionics Integration and System Testing 14-62
Figure 14-15 OT&E Process for Software Maturity 14-64
Figure 14-16 Software Maturity 14-65
Figure 15-1 The Shewhart Cycle 15-7
Figure 15-2 Process Measurement Metrics 15-8
Figure 15-3 Earned-Value through Objectively Observable Milestones 15-13
Figure 15-4 C/SCSC Earned-Value Analysis 15-17
Figure 15-5 Front-loaded Baseline 15-18
Figure 15-6 Rubber Baseline 15-19
Figure 15-7 Effects of Internal Planning 15-19
Figure 15-8 Baseline Budget Exceeds Contract 15-20
Figure 15-9 Requirements and Design Process Metric 15-22
Figure 15-10 Code and Test Progress Metric 15-23
Figure 15-11 Manmonths Progress Metric 15-24
Figure 15-12 Software Quality Assurance and System Life Cycle 15-30
Figure 15-13 Sample Management Structure for an Independent SQA Element 15-31
Figure 15-14 Management Factors to Address for a Quality Product 15-31
Figure 15-15 Defect Rework Hidden Cost 15-33
Figure 15-16 Rework Cost per Development Phase 15-34
Figure 15-17 Defect Causal Analysis Process 15-35
Figure 15-18 F-16 Software Defect Detection Model Results 15-36
Figure 15-19 Industry Average Defect Profile 15-42
Figure 15-20 Defect Profile with Inspections 15-43
Figure 15-21 Personnel Resource Expenditures With and Without Inspections . 15-44
Figure 15-22 Cleanroom Process Model 15-49
Figure 15-23 Cleanroom Pipeline Construction 15-50
Figure 15-24 Cleanroom Certification Process 15-52
Figure 15-25 How Programmers Spend Their Time 15-59
Figure 15-26 IBM's Just-in-Time Training Approach 15-65
Figure 15-27 Space Shuttle Software Development Process 15-67
Figure 15-28 Requirements Tracking for the F-22 15-69
Figure 15-29 Software Configuration Management Process 15-70

Figure 15-30 Iterative Development Products versus Conventional Development Products 15-86
Figure 15-31 Progress Towards Improved Software Economics 15-92
Figure 16-1 Vision for Software 16-5
Figure 16-2 Activity Network Example 16-18
Figure 16-3 Development Time versus Effort Tradeoffs 16-25
Figure 16-4 TI Quality Metrics Benchmarking Summary (1994 Year to Date) .. 16-32

LIST OF TABLES

Table 1-1 GAO Reports on Software Failures 1-8
Table 1-2 Major Non-Military Software Failures 1-10
Table 1-3 Where Military Software Lags Behind Commercial Software 1-32
Table 1-4 Military and Commercial Software Failure Factors 1-33
Table 1-5 Where Military Software Excels 1-35
Table 1-6 Military and Commercial Software Success Factors 1-35
Table 2-1 Software Acquisition Worst Practices 2-33
Table 4-1 SCAI Cost with Megaprogramming 4-55
Table 5-1 Software Engineering Course Length 5-17
Table 5-2 Total SLOC by General Purpose 3GL for Weapons Systems 5-20
Table 5-3 Total SLOC by General Purpose 3GL for MIS 5-20
Table 5-4 New or On-Going Ada Programs Throughout DoD 5-21
Table 5-5 Large-Scale Commercial Ada Systems 5-23
Table 5-6 Industry Experience with Ada 5-37
Table 5-7 FAA Weighted Scores for 6 Criteria Categories 5-38
Table 5-8 SEI Weighted Scores for 6 Criteria Categories (MIS/C3) 5-39
Table 5-9 Productivity Study Comparison (source lines-of-code/manmonth) ... 5-40
Table 5-10 Cost Study Comparison (dollars/source lines-of-code) 5-40
Table 5-11 Integration and FQT Defect Rates 5-41
Table 5-12 Corporate Cost Effectiveness Analysis 5-42
Table 5-13 Comparison of When Ada or a 4GL Is More Cost Effective 5-47
Table 5-14 When Ada Reuse is Most Cost Effective 5-48
Table 5-15 Comparison of Size Impacts Using Three Methods 5-49
Table 5-16 Summary of Ada Program Unit Features 5-56
Table 5-17 Summary of Ada Feature Benefits 5-58
Table 5-18 How to Know When to Transition to Ada 95 5-63
Table 6-1 Top-Ten Risk Identification Checklist 6-23
Table 6-2 Risk Exposure Factors for Satellite Experiment Software 6-24
Table 6-3 Risk Management Questionnaire 6-26
Table 6-4 Team Risk Management Principles 6-28
Table 6-5 Advantages of Team Risk Management 6-28
Table 6-6 Proposal Items or Solution Areas 6-44
Table 6-7 Taxonomy of Software Development Risks 6-45
Table 7-1 BPG Capability Levels, Common Features, and Generic Practices . 7-17
Table 7-2 Summary of SEI CMM℠ Software Process Improvement (SPI) Study 7-30
Table 7-3 OC-ALC Software Process Improvement Benefits 7-34
Table 7-4 Benefits of Moving from Level 1 to Level 3 (SSG Program Example) 7-39
Table 7-5 ROI at Raytheon by Moving from Level 1 to Level 3 7-41
Table 7-6 ROI at Hughes by Moving from Level 2 to Level 3 7-42
Table 7-7 Characteristics of the Complete Dataset 7-56
Table 8-1 Example Management Indicators 8-4
Table 8-2 Collection and Use of Metrics at Loral 8-9
Table 8-3 How Metrics Are Used for Program Management 8-17

TABLE OF CONTENTS

Table 8-4 RADC Software Quality Factors 8-30
Table 8-5 Function Points versus Lines-of-code 8-33
Table 8-6 Function Point Computation 8-35
Table 8-7 Ratios of Feature Points to Function Points 8-37
Table 8-8 Industry Average Productivity Rates (function points produced per manmonth) 8-42
Table 9-1 Impact of Reuse and Moving from Level 1 to Level 3 (113,465 SLOC from reuse) 9-23
Table 9-2 Impact of Reuse at SEI Level 3 (113,465 SLOC from reuse) 9-23
Table 9-3 Quality, Productivity, and Time-to-Market Profiles 9-25
Table 9-4 Relative Cost to Produce and Reuse 9-25
Table 9-5 HP Reuse Program Economic Profiles 9-26
Table 9-6 Traditional versus Mobile Space System Product-Line Approach 9-32
Table 10-1 Countries with the Highest Net Cost per Feature Point Produced 10-4
Table 10-2 Countries with the Lowest Net Cost per Feature Point (US 1991 dollars) 10-5
Table 10-3 Tools to Implement Process Improvement Goals and Remove Management Inadequacies 10-9
Table 10-4 Ten Common Sense Rules for Tool Selection 10-14
Table 10-5 Eight Software Engineering Toolset 10-16
Table 10-6 Tool Selection and Evaluation Activities 10-18
Table 10-7 Common Tools and Methods of a SEE 10-23
Table 10-8 Demonstration Project SEE Tool Suppliers 10-31
Table 10-9 Parametric Models Applicability to Life Cycle Phase 10-41
Table 10-10 Upper CASE Tool Functional Characteristics 10-48
Table 10-11 STSC Test Tool Classifications 10-56
Table 10-12 AdaTEST Features and Functions 10-61
Table 10-13 Program Model Phases and Characteristics 10-89
Table 11-1 Software Supportability Checklist 11-21
Table 12-1 The Software Estimation Process 12-38
Table 14-1 Software Development Life Cycle Cost 14-50
Table 14-2 CMOS Integration Test Results of UT3/Non-UT3 Modules 14-58
Table 14-3 AFOTEC Software OT&E Pamphlets 14-63
Table 14-4 ANSI/IEEE Software Engineering Terminology 14-67
Table 14-5 Military/Commercial Effort Distribution 14-77
Table 14-6 The Ten Commandments of Software Development 14-80
Table 15-1 Flight Plan for Success 15-10
Table 15-2 Software Metrics for Earned-Value 15-25
Table 15-3 1990 US Software Defect Averages (in function points) 15-38
Table 15-4 Fagan Inspection Process Activities 15-45
Table 15-5 Cleanroom Performance Measures (KLOC = 1,000 lines-of-code) 15-54
Table 15-6 Ada Risk Evolution from 1985 to 1994 15-89
Table 16-1 Software Engineering Productivity (1991) 16-11
Table 16-2 US Share of World's Software and Services Market 16-12
Table 16-3 Countries with Highest Software Quality 16-13

ACRONYMS
INDEX

Volume 2

Appendices

PART I
Points of Contact and Web Directories
APPENDIX A Government and Industry Points of Contact
APPENDIX B Web Directory of Software Acquisition and Engineering Policy, Information, Education, and Services

PART II
Policy and Information-Related Appendices
APPENDIX C DoD Policy
APPENDIX D Software-Related Government and Industry Documents
APPENDIX E Selected Technical References
APPENDIX F Selected Reading and Reference Material

PART III
Engineering-Related Appendices
APPENDIX G Software Architecture
APPENDIX H How Should Ada Software Be Documented?
APPENDIX I Comparison of ISO 9001 and CMM[SM]
APPENDIX J Function/Feature Point Counting
APPENDIX K Software Support
APPENDIX L STARS and I-CASE Tools and Services

PART IV
Management-Related Appendices
APPENDIX M Software Source Selection
APPENDIX N AIS ORD Recommendations

PART V
Additional Addenda
APPENDIX O Additional Volume 1 Addenda

PART I
Introduction

Blank page.

CHAPTER

1

Software Acquisition Overview

CHAPTER OVERVIEW

The software industry is reaching its 50 year mark, however, the same problems that plagued us 20 years ago still persist. DoD has had a distressing history of procuring elaborate, high-tech software-intensive weapons that do not work, cannot be relied upon, modified, or maintained. Many of these over budget, overdue programs have been canceled after reaching full-scale production with millions of dollars wasted, and not a single unit reaching the warfighters' hands. With virtually every acquisition snafu, the ***software component*** *can be isolated as the prime source of our dilemmas.*

Over the years, these problems have been studied by experts throughout the industry and Government — all reaching the same conclusion. Our inability to build reliable, economical software is not due to technical shortcomings — but is a product of ***poor management practice****. In this chapter you will learn that software-intensive acquisitions fail due to: (1) the inherent complexity of the software entity, (2) poor estimation of size, time, and cost, (3) unstable requirements, (4) poor decision making by acquisition managers, (5) a belief that something other than improved management skills will cure our ills, and (6) failure to establish and preserve technical in-depth participation and awareness of the state of the program.*

There have been exceptions to the general rule that software-intensive systems are doomed to failure. From these successes we have learned there are certain practices we can apply to our present acquisitions so that, they too, can succeed. The common threads among successful acquisitions consist in the use of improved, modern management techniques by ***both the buyer and supplier****. This chapter discusses using a process-driven acquisition strategy which stresses bringing a supplier on board who has: (1) the most skilled, experienced workforce, (2) a proven track record in developing software of similar size, complexity, and application domain, (3) top-level management commitment to software success, (4) a mature development process and capability, (5) a robust, proven, automated, and appropriately scaled software engineering environment, (6) a well-designed, implementable plan, (7) a defined, established set of standards to guide and control the development effort, and (8) familiarity with the application domain (e.g., there is some 'precedence' knowledge for the application). On the buyer's part, being a good customer means recognizing that successful software is a* ***people thing*** *requiring enlightened leadership based on cooperation, disciplined trust, and inspired teamwork.*

Blank page.

CHAPTER 1

Software Acquisition Overview

SOFTWARE VICTORY: The Exception or The Rule?

In 1987, **Tom DeMarco** and **Timothy Lister**, reported that:

> *Each year since 1977, we have conducted a survey of [software] development projects and their results. We've measured project size, cost, defects, acceleration factors, and success or failure in meeting schedules. We've now accumulated more than 500 project histories, all of them from real-world development efforts...We observe that about 15% of all projects studied came to naught: they were canceled or aborted or "postponed" or they delivered products that were never used. For bigger projects, the odds are even worse. Fully 25% of projects that lasted 25 work-years or more failed to complete...For the overwhelming majority of the bankrupt projects we studied, there was not a single technological issue to explain the failure.* [DeMARCO87]

This chapter tells you why software-intensive acquisition programs fail, and provides you some insight into recognizing major problems while they are still solvable. Thus, we are going to look at an array of major software-intensive acquisitions. The ill-fate of these programs reveals a legacy of buggy software that does not work, can not be trusted, and cannot be modified or maintained. Why are so many software acquisitions delivered late, of such poor quality that they are never used or are even canceled after wasted years and wasted millions?

CHAPTER 1 Software Acquisition Overview

The software industry is reaching its 50 year mark! Over the years we have benefited from the experiences and wisdom of thousands of people reflecting a wealth of knowledge on how to bring a major software-intensive acquisition to successful deployment. Today, there are better management techniques, development tools, and advanced technologies than ever before. Nonetheless, the very same problems that plagued software acquisitions 20 years ago still persist. Why have we neglected to learn from our past failures, and albeit rare, successes? What you are going to learn throughout these Guidelines is that the success or failure of a software acquisition is a people thing — not a tool or technology thing or any other kind of thing. Granted, most successful development organizations take full advantage of state-of-the-art technologies. But acquisition programs are led by people, not technology. [WHITTEN95]

It is no mystery why software acquisitions get into trouble. Enough has been written on the subject to overload a C-17 Globemaster. Poor management is cited time and again as the nemesis. If we know this, why do we keep making the same mistakes over and over? Because we do not listen, and we do not act on what we have learned! For us to conquer the software war, we must win the management battle. If we do not re-engineer the software management task, we are doomed to join the ranks of the software acquisition norm — program failure! Our purpose is to provide you with the ability to recognize major problems while they are still solvable. Once you understand what these problems are and why they occur, you will be equipped to do something about containing and/or eliminating them.

Remember, learning from past mistakes must be accompanied with a sincere commitment to better management practices. It must also come from proactive leadership to follow your convictions, to demonstrate the courage and integrity to take charge, and to strive to stay in control. As **Lloyd K. Mosemann, II**, former Deputy Assistant Secretary of the Air Force, (communications, computers, and support), told a large audience of software professionals at the 1995 Software Technology Conference in Salt Lake City, "*I'll stop preaching, when you start practicing!*" What we preach here is founded on valuable, hard-learned lessons, and a wealth of knowledge gathered from experts and learned practitioners throughout the software community. Do not let your program become another relic on the crowded battlefield of software defeats. There are solutions. Read, absorb, and act upon what you learn in these Guidelines — ***then start practicing!***

SOFTWARE: The Highest Risk System Component

Historically, the Department of Defense (DoD) has been plagued with many well-publicized acquisition snafus. The situation was summarized in the following 1985 *Business Week* article.

> *When the Pentagon decides to build a complex new weapon these days, it often seems to run into disaster. The promise of advanced technology seduces designers and eager contractors into taking big risks with the public's money. And frequently, these elaborate projects end up hamstrung by technical errors, management miscalculations, or congressional interference. The result is weapons that are grossly overpriced — or don't work.* [BUSWEEK85]

A decade later, problems still persist. On April 27, 1995 **Frank C. Conahan**, Senior Defense and International Affairs Advisor to the Comptroller General of the United States, testified before the Committee on the Budget, US House of Representatives. In his statement he explained,

> *Over the years, we have reported on the persistent problems that have plagued weapons acquisition. Many new weapons cost more, are less capable than anticipated, and experience schedule delays. These problems are typical of DoD's history of inadequate requirements determinations for weapon systems; projecting unrealistic cost, schedule, and performance estimates; developing and producing weapons concurrently; and committing weapon systems to production before adequate testing has been completed.* [CONAHAN95]

Why is DoD still plagued with software-intensive acquisitions gone astray? When a major procurement program turns into a fiasco, when costs soar, deliveries fall behind schedule, and performance is compromised, more times than not the problem can be traced to one high-risk component — the software! In December 1990 a series of articles ran in *The Washington Post* explaining that:

> *Software problems have caused major delays of weapons systems, created malfunctioning aircraft, and cost the Defense Department billions of dollars in unanticipated costs. Officials acknowledge that virtually every troubled weapon system, from the electronics of the B-1B bomber to satellite tracking systems, has been affected with software problems. Even straightforward record-keeping systems can get bogged down; last year the Navy canceled a software accounting project nine years in the making after its cost quadrupled to $230 million.*
> [RICHARDS90[1]]

In the same series **Colonel Joseph Greene, Jr.** (USAF), head of a Pentagon software research effort, was interviewed about a study he completed on 82 large military procurement programs. Of those, Greene found that programs relying heavily on software ran 20 months behind schedule—3 times longer than non-software-intensive programs. He calculated that those delays cost DoD 1/10th of its $100 billion research and procurement budget. Greene explained, "The department is paying a huge penalty for not dealing with its software problems. The penalty is not just late software — it is degraded war-fighting capability." [GREENE90] (In another study *The Washington Post* cited, 3/4ths of 55 aerospace and defense contractors ran their software programs in an ad hoc, chaotic manner.) [RICHARDS90[2]]

Often taken for granted in the past, software is now recognized as the highest risk system component in virtually every software-intensive acquisition. A June 1991 report of the **Defense Acquisition Board** claimed that the estimated 1.55 million lines-of-code to be built for the **F-22** — the largest software task ever undertaken for an attack/fighter program — represents the most significant threat to the successful development of the aircraft. In October 1994, F-22 program managers concurred that the avionics software integration is the most formidable task for their contractors — the accomplishment of which the successful deployment of the F-22 depends (budgetary issues not considered). [GAO95[1]]

CHAPTER 1 Software Acquisition Overview

False Steps on the Battlefield

> *[There is] room for a military criticism as well as a place for a little ridicule upon some famous transactions...But why this censure... to exercise the faculty of judging...* ***The more a soldier thinks of the false steps of those that are gone before, the more likely he is to avoid them.***
> — Major General Sir James Wolfe [WOLFE27]

Will studying the false steps of past software-intensive acquisitions give us insight into how not to make the same mistakes over again? Can we learn from past failures so we might proceed on firmer ground? The following discussion examines the relics of some famous program defeats that litter the software battlefield.

General Accounting Office (GAO) Reports

DoD is not alone in rampant software acquisition debacles. In 1979, the GAO (the Congress' watchdog agency) published the report, *Contracting for Computer Software Development — Serious Problems Require Management Attention to Avoid Wasting Additional Millions*. The results of that report and selected others over the years are listed on Table 1-1 (below).

The 1979 GAO report made some stirring comments about the state of software development in the late 1970s, which unfortunately concur with more recent observations. It stated:

> *Several factors contributed to the situation [of having software development problems]. First, the invisible nature of both the work process and its product made software projects very difficult to manage and predict. Second, the explosive growth of the use of computers created demand for new programmers, most of whom were self-taught on the job; and frequently, low productivity and poor quality resulted. Third, there was little idea then of how to train programmers properly. Fourth, a tradition grew that programmers were secretive craftspersons, whose products, during development, were their own property.* [GAO79]

GAO REPORT	REPORT FINDINGS
Contracting for Computer Software Development: Serious Problems Require Management Attention to Avoid Wasting Additional Millions November 9, 1979 (FGMSD-80-4)	Analysis of custom-built MIS systems (163 contractors and 113 government personnel surveyed) produced the following results: • + 60% of contracts had schedule overruns • + 50% of contracts had cost overruns • + 45% of software could not be used • + 29% of software was never delivered • + 19% of software had to be reworked to be used • - 3% of software had to be modified to be used • - 2% of software was unusable as delivered
Sergeant York: Concerns About the Army's Accelerated Acquisition Strategy May 1986 (GAO/NSIAD-86-89)	• 64 (of planned 614) units delivered and subsequently scrapped • FOT&E results showed significant performance shortfalls • Cost and schedule overruns projected if government demanded required functionality • $1.8 billion lost • **Program canceled**
Navy Decision to Terminate Its Standard Automated Financial System March 1989 (GAO/IMTEC-89-37)	• $446.5 million (99.9%) projected cost overrun • 5 year projected schedule overrun • $230 million lost • **Program canceled**
Embedded Computer Systems: Significant Software Problems on C-17 Must Be Addressed May 1992 (GAO/IMTEC-92-48)	• 2 years behind schedule (as of March 1992) • $1.5 billion cost overrun • Software size/complexity underestimated • MilStds waived for contractor with limited software experience • Shortcuts taken on software testing and software supportability issues
Software Challenges in Mission-Critical DoD Systems December 24, 1992 (GAO/IMTEC-93-13)	15 major systems studied had the following common problems: • Poor software engineering concepts, methods, and practices used • Proceeded despite serious problems • Requirements were ill-defined and unstable • Architectures were inflexible • Security requirements not met • Poor testing methods and procedures used • No system-level integration testing performed

Table 1-1 GAO Reports on Software Failures

GAO REPORT	REPORT FINDINGS
Air Traffic Control: Advanced Automation System (AAS) Problems Need to Be Addressed March 10, 1993 (GAO/T-RCED-93-15)	• 5-years behind schedule • $2.6 billion cost overrun • $238 million spent due to delays
Attack Warning: Status of the Cheyenne Mountain Upgrade Program September 1994 (GAO/AIMD-94-175)	• 8 years behind schedule (at time of report) • $792 million over budget (at time of report) • 11 years projected schedule slip • $896 million projected budget overrun • $22 million/year additional costs for continued operation/maintenance of old system
New Denver Airport Impact of the Delayed Baggage System October 1994 (GAO/RCED-95-35BR)	Denver International Airport (DIA) opening delayed by 2 years • $27 million cost overrun on baggage system • $55 million spent by United Airlines to fix their portion of baggage system • $51 million spent by City of Denver to cannibalize and haul manual system from old airport to new for other airlines • $35 million projected to expand repaired automated system for airlines other than United • $360 million total delay costs to DIA and airlines • $37 million in DIA lost income • $8 million in bond fees • $20 user fee per enplaned DIA passenger tacked on to ticket price
Comanche Helicopter: Testing Needs to be Completed Prior to Production Decisions May 1995 (GAO/NSIAD-95-112)	• Cost tripled in 10 years (from $12.1 million in 1985 to $34.4 million in 1995, 185% cost increase) • Software development and testing problems • Required performance has been decreased by 74%
Air Traffic Control: Status of FAA's Modernization Program May 1995 (GAO/RCED-95-175FS)	Advanced Automation System (AAS) restructured • $5.6 billion cost overrun • 8 year schedule slip • 2 of 4 systems canceled; 3rd system reduced by 48%; restructured cost $6 billion • Required capacity downgraded

Table 1-1 GAO Reports on Software Failures (cont.)

Scientific American Article

Software acquisition failures are also common place in the private sector. "Software's Chronic Crisis," an article from the October 1994 *Scientific American, [see Foreword]* cites the results of an IBM study of 24 leading companies developing large, distributed software-intensive systems. Of all those companies' software programs combined,

- **55%** had cost overruns;
- **68%** had schedule overruns; and
- **88%** had to be redesigned to be used.

According to the article, studies show that:

- **1/3rd** of all large-scale software programs are canceled;
- The average software program overruns its schedule by **50%** — larger programs usually by more;
- **3/4ths** of all large-scale developments are *operational failures* — they do not function as intended or are not used at all; and
- Software is still hand-crafted by artisans using techniques they can neither measure nor consistently repeat. [GIBBS94]

Table 1-2 lists the major software development failures discussed in the *Scientific American* article.

YEAR	PROJECT	RESULTS
1980s	International Telegraph & Telephone (ITT) *4 switching systems*	• 40,000 function point system • $500 million lost • **Canceled**
1987	California Department of Motor Vehicles *Automated Vehicle/ Driver's License System*	• 3 (5,000 function point size) switches • $30 million lost • **Canceled**
1989	State of Washington *Automated Social Service Caseworker System*	• 7 years to build • Failed to meet user needs • $20 million lost • **Canceled**
1992	American Airlines *Flight Booking System*	• $165 million lost • **Canceled**

Table 1-2 Major Non-Military Software Failures

Battle Damage Assessments

Over the years, our software problems have been analyzed by the most experienced and savvy experts on the subject. The **Defense Science Boards** and **Process Action Teams** who wrote the reports and articles discussed below were comprised of the best and brightest software engineers from Government, industry, and academia. Their analyses of our problems, suggested resolutions, and insights into implementing solutions are as astute and knowledgeable a disposition of the software condition as can be found. As you read the following summaries, which span almost two decades (starting with the 1979 GAO report above), a startling paradox will emerge. You will see there is little deviation from the common perception that one isolated factor continues to surface as the primary cause of our software problems — ***poor management***. The paradox is that while we have been told over and over our problems lie in management — we have made little progress in correcting our management shortfalls. Major software developments blossom into fiascoes which end in disasters with surprising constancy. The problems that were present in 1979 — that have showed their ugly heads in subsequent programs — still trouble us today. The paradox is also compounded by the fact that the development of software has become increasingly more demanding and complex. As William Wulf, former head of the **National Science Foundation**'s office of computer and information science, warned in 1990, *"The amount and quality of software we need is increasing constantly, and our ability to produce it is essentially stagnant. Those two things are on a collision course...[It is] absolutely a problem of much larger dimension than most people realize."* [WULF90]

The Parnas Papers

September 1985 marked the resignation of **David Parnas**, a leading software engineer of the time, from the **Panel on Computing in Support of Battle Management**, convened by the **Strategic Defense Initiative (SDI) Organization (SDIO).** In his letter of resignation, he included eight essays (published in *American Scientist*) detailing why he believed SDI software problems were insurmountable — that the software conceived for the system ***could not be built***. The SDI was a massive software network linking an equally huge array of sensors and weapon systems — all software-intensive in their own right. It was to identify, track, and direct defensive weapons towards

space-borne targets, the ballistic characteristics of which would not be known with any certainty until the moment of engagement. Because it was to defend against incoming nuclear warheads, operational testing of the system under real-time battle conditions would not be possible prior to its deployment. Also, because the system's service life would be so short (30-90 minutes), there would be no time to debug or modify its mission-critical software. In his analysis of why SDI software could not be deployed with any degree of reliability, Parnas conveyed some crucial insights into the state of software development practice. Parnas told us that software is risky because, ***"Software is hard"*** [to build]. Often described as the most tasking mental activity ever undertaken by mankind, software is hard because it is so ***complex***. He explained that:

- *Software development is a trial-and-error craft. People write code without any expectation that they will be right the first time. They spend at least as much time testing and correcting errors as they spend writing the initial code.*
- *The military software we depend on every day is not likely to be correct. The methods in use in the industry today are not adequate for building large, real-time software systems that must be reliable when first used.*
- *Good software engineering is far from easy. The methods reduce, but do not eliminate, errors. They reduce, but do not eliminate, the need for testing.*
- *It is clear that the [tool] environment in which we work does make a difference. The flexibility of the [environment] allows us to eliminate many of the time-consuming housekeeping tasks involved in producing large systems.*
- *Even with sound software design principles, we need broad experience with similar systems to design good, reliable software.* [PARNAS85]

Of course, Parnas made his observations in the context of the most colossal real-time software system ever conceived—the SDI. But his judgment about the immaturity of software development applies to the discipline as a whole and to its inability to successfully produce any large-scale software system. Parnas did not give credence to optimistic claims of future breakthroughs in software technologies or practices of the day. He claimed they could not promise improvements

significant enough to allow the SDI to be built as conceived. In fact, he called some of them *"dangerous and misleading."* He also said major achievements in software quality and productivity would not come from any of the areas he examined in his essays (e.g., languages and tools, automatic programming methods, formal software verification, artificial intelligence, and software research). Parnas claimed that because software is built by trial and error, ***the only reliable software that can be built is that which is precedented.*** *[NOTE: "Precedented," as used here, refers to software which is based upon an existing model or example.]* This is the fundamental motivation for advanced process models based on iterative development.

Fred Brooks' *"No Silver Bullet"*

The April 1987 issue of *Computer* included the most famous and oft quoted paper on the software dilemma ever published, "No Silver Bullet: Essence and Accidents of Software Engineering," by software pioneer **Frederick P. Brooks, Jr**. In it he likened our monumental problems with software to werewolves. Like werewolves, software programs can unexpectedly be transformed from *"commonplace"* into untamable monsters. A normal, routine software development has the capability of becoming — without warning — a disaster marked by blown schedules, busted budgets, and bad products. Whereas, a **Silver Bullet** has the power to slay a werewolf, Brooks claimed there exists no Silver Bullet that can magically slay our software problems.

Brooks saw the difficulties facing improvements in software technology as characterized by two major obstacles. One is the *essence* of software — the difficulties inherent in the software beast itself. The other is the *accidents* — those difficulties found in the production of software that are not inherent to the beast. Of the essence, Brooks, like Parnas, took the position that ***software is hard*** and always will be. It has an inherent and necessary complexity. *"Software entities are more complex... than perhaps any other human construct...Software systems have orders-of-magnitude more states than computers do...[Because] the complexity of software is an essential property,"* it does not lend itself to simplification techniques found in other disciplines. [BROOKS87] For instance, in mathematics simplified models of complex problems are often used as analytical tools. The essence of software is that it

achieves the solution of a complex problem by compounding its complexity (i.e., the algorithms defining the solution are more complicated than the real-world problems they solve.) *[See Chapter 5, Ada: The Enabling Technology, for a discussion on problem and solution domains.]* [GLASS91]

Also like Parnas, Brooks analyzed the major breakthroughs in software development which have increased productivity and improved quality over the years. Advances, such as high-order languages, faster processing times, and integrated software engineering tool environments, have achieved quantum leaps in dealing with the accidents. But he, too, concluded that it is doubtful further technological advances of any magnitude will solve our chronic problems. Huge promises of Silver Bullets that will yield spectacular progress in software development (common occurrences in the hardware arena) are not to be believed.

The solutions to the software dilemma, Brooks suggests, are not as colorful as his description of the problem. They are, in fact, rather pedestrian and mundane. Brooks tells us *"a disciplined, consistent effort to develop, propagate, and exploit the following suggestions should yield an order of magnitude improvement."*

- **Buy software, rather than build it.** *"Every day this becomes easier, as more and more vendors offer more and better software products for a dizzying variety of applications." [Commercial-off-the-shelf (COTS) software is discussed in Chapter 13, Contracting for Success.]*
- **Grow software, don't build it.** Develop software incrementally and refine requirements through prototyping. Partial solutions are easier to correct and modify than a full-blown, finished product that does not perform as envisioned. *[Incremental development is discussed in Chapter 3, System Life Cycle and Methodologies; prototyping is discussed in Chapter 14, Managing Software Development.]*
- **Employ and cultivate the best and the brightest.** *"Sound methodology can empower and liberate the creative mind; it cannot inflame or inspire the drudge. Great designs come from great designers!"* [BROOKS87] *["Smart People" are discussed below.]*

1987 Defense Science Board Report on Military Software

Also in 1987, a final report was released by the **Defense Science Board Task Force on Military Software**. The Task Force was chaired by **Fred Brooks** and manned by some of the most diligent, astute experts in the field. It pulled no punches in waging a frontal attack on DoD's on-going software troubles when it stated:

- *Many previous studies have provided an abundance of valid conclusions and detailed recommendations.* ***Most remain unimplemented.*** *If the military software problem is real, it is not perceived as urgent.*
- *We do not see any single technological development in the next decade that promises ten-fold improvement in software productivity, reliability, and timeliness.*
- *Few fields have so large a gap between best current practice and average current practice.*
- *The Task Force is convinced that today's major problems with military software development are not technical problems, but* ***management problems.*** [DSB87]

The report assaulted not only the institutions that govern military software development, but the obstacles encountered when transitioning technology and modern management practices to a new engineering discipline. The report's recommendations focusing on the software development task are summarized as:

- *DoD should assume software requirements can be met with* ***COTS*** *subsystems and components until it is proved they are unique [requirements]. [Discussed in Chapter 13, Contracting for Success.]*
- *DoD management should commit to a serious and determined push to* ***Ada****. [Discussed in Chapter 5, Ada the Enabling Technology.]*
- *DoD should develop* ***metrics*** *and* ***measuring techniques*** *for software quality and completeness, and incorporate these routinely in contracts. [Discussed in Chapter 8, Measurement and Metrics.]*

- *DoD should examine and revise regulations to approach modern **commercial practice** insofar as practicable and appropriate. [Discussed in Chapter 2, DoD Software Acquisition Environment.*

NOTE: Only commercial practices which enhance the common thread attributes of successful programs referenced above should be adopted. Avoid all others.

- *DoD should mandate the iterative setting of specifications, the **rapid prototyping** of specified systems, and **incremental development**. [Discussed in Chapter 3, System Life Cycle and Methodologies.]*
- *DoD should mandate the use of **risk management** techniques in software acquisition. [Discussed in Chapter 6, Risk Management.]*
- *DoD should develop economic incentives for contractors to offer modules for **reuse** and to **buy** modules rather than building new ones. [Discussed in Chapter 9, Reuse and Chapter 13, Contracting for Success.]*
- *DoD should enhance **education** for software personnel. [Discussed in Chapter 15, Managing for Process Improvement.]*

The thrust of the report is summarized in the following statement: *"We call for no new initiatives in the development of technology, some modest shift of focus in technology efforts under way, but major re-examination and change of attitudes, policies, and practices concerning software acquisition."* [DSB87]

1992 Software Process Action Team Report

In June of 1992, the Air Force Systems Command **Software Process Action Team** published their final report, ***Process Improvement for Systems/Software Acquisition***. Using a rigorous **Total Quality Management (TQM)** approach, the team focused their analysis on software-intensive acquisition programs where the software component could be cited as the primary contributor to, if not the main cause of, cost, schedule, and performance violations. They identified the three top problem areas as: (1) program acquisition baselines, (2)

requirements, and (3) program management. The report summarized problems, provided solutions that included process changes, and made recommendations for implementing those solutions. Recommendations included organizational changes, changes to standards, regulations, and other implementation vehicles, improved training, use of better metrics, improved funding estimates, adhering to current *best practices,* and identifying outstanding risk areas.

- **Program acquisition baselines**. Not enough schedule time is allotted to develop the necessary software. Baselines are established prematurely, and once set, there is reluctance to change them. Baselines are frozen before system and software requirements have been defined or derived and before there is an adequate basis for cost and schedule estimation, especially with unprecedented systems. ***Recommendations***: (1) formal methodologies, (2) phased development, and (3) bidding to an open schedule.
- **Requirements**. Often, there is a failure to match user needs with government/industry resources and to define and follow an adequate systems engineering process. There is also a lack of clear, well-defined requirements. ***Recommendations:*** (1) user involvement in requirements definition and on-going communication among the acquisition, logistics, developer, tester, and user communities; (2) a clearly-defined systems engineering process; and (3) an improved specification process with focus on requirements allocation, derivation, and traceability.
- **Program management**. Technical reviews are improperly driven by management considerations and software documentation requirements are often excessive in volume and poor in quality. ***Recommendations***: (1) enhancement of the technical review process for software related milestones; (2) more flexible data items to improve software documentation; (3) system acquisition training programs with more focus on software; (3) better definition of the systems engineering process with respect to the software engineering process; and (4) incorporating incremental development in software management standards and data requirements. [PAT92]

1994 Defense Science Board Report on Acquiring Defense Software Commercially

The most recent report is that of the 1994 **Defense Science Board** which makes a pertinent point about the trend (or lack of trend) in DoD software acquisition when it states: *"Despite the increased emphasis given to software issues by the DoD...the majority of the recommendations resulting from these studies have* ***not been***

implemented." [DSB94] Some of the report's recommendations germane to this discussion are listed here. *[Other recommendations are discussed in Chapter 13, Contracting for Success.]*

- Establish mechanisms to allow both current ability to perform and past performance as key factors in source selection;
- Define software architectures to enable rapid changes and reuse;
- Facilitate early system engineering and iterative development;
- Require program managers to stay with programs at least through beta testing to maintain continuity and understanding of original requirement nuances.

WHY SOFTWARE ACQUISITIONS FAIL

Software acquisitions fail because software management fails! Software management fails in three areas: administration, program measurement, and technical scrutiny. In Chapter 7, *Software Development Maturity*, you will learn about a method for determining the maturity of contractors and in-house software development organizations. Ratings go from a Level 1, *Initial*, to a Level 5, *Optimizing*, with levels of increasing maturity in between. Our management problems can often be blamed on the fact that our development processes are immature and chaotic. Our development costs and schedules, as well as product quality, are unpredictable. ***Our managers are poorly trained because the software development process is misunderstood.*** The maturity of the processes within a software organization is a very important determinant for software success. However, success is also dependent on the ***people*** within the organization and their ability to technically evaluate their software product.

Our quality problems quickly become exorbitant cost problems. Once the software is delivered and in the user's hands, latent defects often need correction. As with most systems that have long operational lives, software must be modified to adapt to new or changing requirements and/or upgraded to new technologies. Before DoD's mandate to use the Ada language, over 400 different languages were used to develop DoD software. Our software was often so unique and custom-crafted, and so poorly documented, the only one who could figure it out was the original designer — who was often off working on something else or otherwise unavailable. Phenomenal cost growths occur once the software enters its operational life. This is manifest in

software that imposes heavy training loads, increases labor expenditures, and frustrates attempts at reusing software developed for one system in another.

Blum sees the software development process from three perspectives. He describes software design as looking forward, software quality assurance as looking backward, and management as looking downward. Although not earthshaking, his explanation does place management above the detailed, technical work — looking down. This, he claims, makes the management task the most arduous in software development — and consequently, the major source of all our problems. [BLUM92] In his book, The Five Pillars of TQM, **General Bill Creech** (USAF retired) makes the same observation:

> *You must do more than talk about it; you must change the organization 'conceptually' and 'structurally' to bring leadership alive at all levels. Principles flow from the top down; decisions flow from the bottom up.* [CREECH94]

In other words, management fails because decision making is coming from the wrong direction. We are trying to make decisions from the top down, when they need to come from the bottom up. To control it, ***managers need a better understanding of the software beast.*** Overwhelming evidence indicates that software programs fail for the following common reasons:

- Software's inherent complexity (non-management related);
- Our inability to estimate cost, schedule, and size;
- Unstable requirements; and
- Poor problem-solving/decision-making.

No quick, easy solutions exist for these major, recurring, oft-repeated problems. *If there were, these Guidelines would not be so thick!* The battle damage assessments in the previous section illustrate the severity of our problems and the range of solutions. As you will see, solutions are often interrelated and multifaceted. Those software best practices you should implement to avoid or correct these problems are discussed throughout these Guidelines.

NOTE: If your program is experiencing any of the following problems, it is suggested you read these entire Guidelines, and especially Chapter 16, *The Challenge*, which discusses "*What to Do with a Troubled Program.*"

Complexity

As **Parnas** and **Brooks** explained, software is risky because it is ***hard*** to build. The complexity of hardware pales in comparison with the complexity of software. For any given hardware problem, there is usually a high percent of component reuse and a finite number of solutions. With software, even with optimized solutions, there are a near-infinite number of possible correct solutions. Theoretically, any set of problems from any other discipline can be solved within the software domain. [GLASS92]

Complexity plagues us because we often fail to take a disciplined approach to design and create more complexity than needed. Although sometimes necessary to match problem complexity, software complexity should be kept to a minimum by all means. ***Highly complex solutions are destined to be high cost maintenance nightmares!*** The more complex the software — the more difficult it is to understand — the greater the chance for defects to propagate throughout the code. The cost of making changes and correcting defects often soars beyond acceptable levels, resulting in programs abandoned after exorbitant expenditures of unrecoupable resources. As **Brooks** tells us, the best designs are those that *"produce structures that are faster, smaller, simpler, cleaner, and produced with less effort."* [BROOKS87]

Inadequate Estimates

The fundamental reason software-intensive developments overrun cost and schedule, with resulting quality and performance shortfalls, is our ***inability to estimate***. No matter how smooth our development process, how efficient our tools, or how smart our designers, our predictions of cost and schedule are frequently out of sync with what actually occurs in the production of a software product. We often forget that software development involves much more than simply writing code. For example, we are still learning that software inspections and testing take longer than anticipated and that maintenance consumes from **60%** to **80%** of our software dollars. We also do not account for the amount of ***scrap*** and ***rework*** of code involved when a developer has an *ad hoc*, chaotic development process, the cost of which Boehm claims to be about **44%** of every dollar spent, as illustrated in Figure 1-1. [BOEHM81] *[Rework is discussed in Chapter 8, Measurement and Metrics.]*

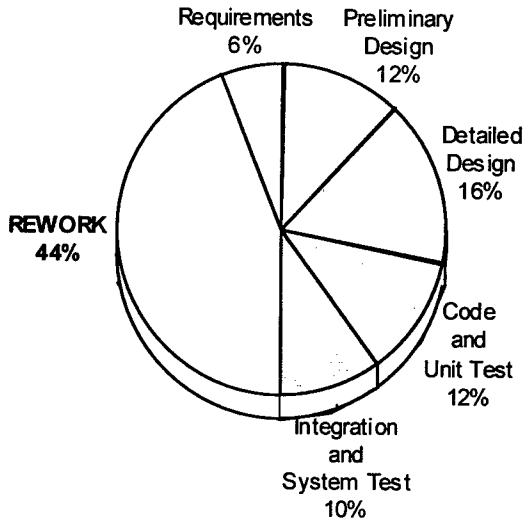


Figure 1-1 Defect Rework Hidden Cost [BOEHM81]

Size and Complexity Estimates

Predicting the **size** and **complexity** of the software to be built is at the heart of our estimation deficiencies. Because software is intangible — we cannot weigh it, box it, put our arms around it, or paint a picture of it — our attempts at projecting how big it will be, how hard it will be to create, or how long it will take are compounded. When a software development is precedented (i.e., a similar system has been developed) size and complexity projections (and thus cost and schedule) are usually more accurate. In unprecedented systems, however, our inability to estimate the intangible is acute. *[See Chapter 8, Measurement and Metrics, and Chapter 12, Strategic Planning, for discussions on software estimation and planning.]*

Cost/Schedule Estimates

Programs tend to get in trouble in small, progressively compounding increments. When the product is late (and/or over cost), we apply management pressure to reduce the slack between our projected delivery date and the illusive real one. This aggravates the problem into a *Catch-22* situation. With inadequate resource and schedule estimates, the time required to *build-quality-in* may be insufficient. Also to meet schedule and keep down cost, the next easiest thing to cut is testing. [GLASS92] Before we realize it, a late, over cost program

evolves into an unreliable one. From the developer's point of view, when a cost/schedule disaster is discovered, they often try to protect their contract through alternative proposals that attempt to deliver less for the same price. This leads to *down-scoping,* or eliminating requirements, in an attempt to stay within initial projections. [MARCINIAK90] This is a very serious situation because it means resources have been expended, often exhausted, and the user does not get the system for which they paid. When this happens, we have the *"perfect formula for a software disaster!"* There are many cases of programs that have been canceled without the delivery of a single operational product after years of schedule and cost overruns.

Schedule changes, due to unrealistic estimates of required development time, often start as unnoticeable alterations in plans that go undetected by most managers. **Schedule slips**, starting small, have the potential of becoming major problems because even small slips impact delivery of other related elements and almost always affect cost. Late software (on the system's critical path) causes other system components to slip their schedules while waiting for delinquent software. Failing to recognize and deal with this problem through expeditious corrective action, the situation can quickly deteriorate into another *"perfect formula for a software disaster!"*

Optimistic Estimates

In DoD we are subject to spending and budgeting scrutiny from the Congress, the press, and upper management. Under pressure, contractors and military managers often make overly **optimistic estimates** about how much the software will cost and how long it will take to produce. We often discard pessimistic cost, schedule, and size estimates and base our projections on the *best of all possible worlds.* We fail to manage risk and to build a *management reserve* or a *worst-case scenario* into our cost/schedules for fear our programs will not get funded or approved if we submit more realistic figures. This increases the likelihood for *"perfect formula for a software disaster"* shortcuts in the development of a product that was improperly funded and scheduled. The problem, in these cases, is not the actual estimates of cost, schedule, and size (which in many cases may be reasonable estimates) — it is the failure to use these estimates to establish reasonable, attainable program baselines.

Unstable Requirements

One big cause of software program failures, upon which all the reports and studies undeniably concur, is **requirements instability**. Because user missions evolve as the world and threats evolve, it is reasonable to assume requirements are going to change. Thus, the first factor to consider when managing unstable requirements is to build software systems with an architectural that tolerates changing requirements. It is important, however, not to compromise the overall architectural design for a subtle near-term requirement, that if compromised, will negate the ability for future change. The second factor to consider is to control how and at what pace inevitable requirements changes are incorporated. If *ad hoc*, sporadic, or frequent modifications to requirements or their interpretation are inflicted on developers, ***creeping changes in cost and schedule*** are a given. Sometimes what appear to be minor changes have dramatic *side-effects* elsewhere in the software. If full (technical and effort) evaluation of change consequences are not included in the management process, the incremental incorporation of changed requirements can invalidate estimates of cost and schedule — diminishing product quality.

One potential source of instability is inadequately stated requirements. Indefinite and undefined software requirements also lead to creep cost and schedule changes which can continue even after the program enters development. The most important, yet difficult, software development task, **requirements definition** and **analysis**, plagues software programs in all domains — in all sectors of the industry. In DoD this is especially menacing, where if poorly executed, ill-defined requirements lead to poor specifications which impact cost, schedule, and latent defect rates. Requirements creep has the greatest impact on our inability to estimate. Figure 1-2 (below) illustrates how the ability to predict software cost increases as requirements become progressively better defined. [BOEHM81]

User Involvement

Paul Paulson, president of Doyle, Dane and Bernbach, a large New York brokerage firm, was quoted in the *New York Times* as saying,

> *You can learn a lot from the client. Some 70% doesn't matter, but that 30% will kill you.* [PAULSON79]

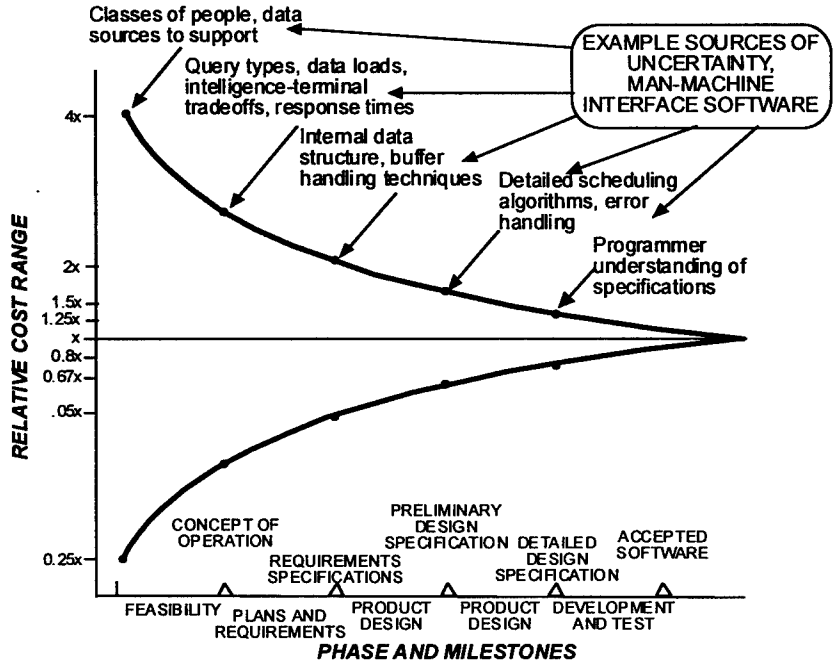


Figure 1-2 Software Cost Estimation Accuracy versus Phase [BOEHM81]

Misinterpretation of user requirements is a major, if not the greatest, contributor to software failure. Not understanding your client (the military user) while managing software development is one sure way to make your program *crash-and-burn*. Inadequately stated requirements have often been the source of costly support problems and ultimate program failures.

User involvement is critical throughout requirements analysis and design, where user feedback is essential to determine whether perceived user needs have been correctly translated into software functionality. The 1992 final report of the **Software Process Action Team** *[discussed above]* found that the primary reason for major software-intensive program failures was the ***inability to translate user needs into viable software requirements***. The report states:

> *The procurement process often results in government acquisitions that fail to meet user needs. The problem is exacerbated during system development when requirements decisions are made without* ***adequate user***

> ***input*** *and without full understanding of the overall impact on costs, schedules, performance, and other critical factors. Current Government and industry practices have led to requirements specifications that contain design information, inappropriate levels of detail, inadequate requirements, and poor traceability.*

Lessons-learned from the Air Force's **Nuclear Mission Planning and Production System (NMPPS)** warn that users often do not start out with a *clean slate* when explaining their operational, readiness, and logistics requirements. Additionally, they may view the statement of their requirements as one more document being coordinated within headquarters which can be changed with minimal impact. Personnel with operational experience can also contribute to this paradox because they, too, assume they generally know the requirements, and need to ask users fewer questions. [KEENE91]

Another example of the user involvement issue occurs when a typical program office designates someone other than the user with responsibility for defining system requirements. These *user representatives*, often called functional analysts, use a systems analysis approach to functional design development. While some functional analysts have extensive background in the target system, others rely on their understanding of user requirements. In both cases, understanding user requirements quickly diminishes without frequent exposure to the target system's operational environment.

Once developed, functional specifications are passed on to programmers who must interpret specifications and write the code. Large programs can have 50 or more programmers receiving functional guidance using this methodology. Considering that initial guidance is likely to be, at best, partially flawed, a second translation compounds the situation. During system testing, efforts are expended in determining whether a software error was introduced during functional design or in actual coding. The likely result of this design-to-product process is the most costly *perfect formula for a software disaster — **software redesign***. [HENDERSON95]

Communication

The main reason errors occur during requirements definition and analysis is lack of **communication**. During the requirements phase, the user tries to articulate a concept of the system function and performance into concrete detail. The software engineer attempts to translate user definitions into models of required information, control flow, operational behavior, and data content. The chances for misinterpretation, misinformation, and ambiguity abound. **General John W. Vessey**, when serving as Chairman of the Joint Chiefs of Staff, explained that

> *More has been screwed up on the battlefield and misunderstood at the Pentagon because of the lack of understanding of the English language than any other single factor.* [VESSEY84]

Lack of understanding of what software is, how it performs, and our difficulty in conveying what we want it to do, are compounded by the inherent shortcomings of the English language. The dilemmas confronting software engineers are expressed in the statement by the user: *"I know you believe you understood what you think I said, but I'm not sure you realize that what you heard is not what I meant."* [PRESSMAN92] Under these conditions, designers fail to translate conceptual user needs into functional software requirements. Software that does not fit the needs of the customer is doomed for the trashheap! As Brooks explains,

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.* [BROOKS87]

Data collected at **Rome Laboratory** indicate that over ***50% of all software errors are "requirements errors."*** Requirements errors are more expensive to correct the further they percolate throughout the life cycle. ***It is often 50 times more expensive to correct a defect during systems integration than during requirements analysis.*** [DiNITTO92] Figure 1-3 illustrates how the cost to correct requirements errors increases during subsequent phases of development.

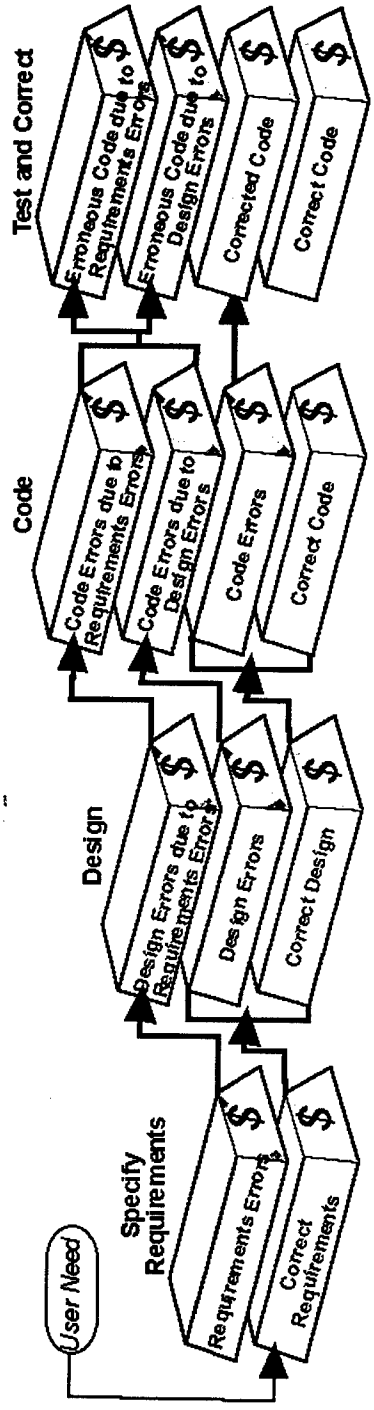


Figure 1-3 Error Propagation Cost

Intangibility

The reasons we have trouble estimating how big, how complex, or how long it takes to produce software are the same reasons that confound us in determining what and how the software should perform. Software can be designed, but it cannot be built in any physical sense. Users have trouble describing something in English that is invisible, exists in an ethereal world of magnetic fields, and communicates in electronic bits and beeps.

Complexity

The **complexity** issue arises again when confronted with the enormous tasks we often want our software to perform. Using the SDI as an example, we undertake giant, unique developments that require years of work, and hundreds of people to produce. Being unprecedented, they cannot be built using previous knowledge and cannot be tested under actual operational conditions or in the environment in which they will be used. Nailing down how the software should perform under these circumstances is often problematic. *[See Chapter 8, Measurement and Metrics, for a discussion on measuring complexity.]*

Changing Threat

Military operations constantly change in response to volatile world events, often causing the mission to change midstream during development and almost routinely after deployment. These changes in world events can cause requirements instability that must be addressed in the most cost-effective and easiest manner. Because ***software is soft*** and flexible, we choose to change it rather than bend metal. For example, in response to the changing world threat, the B-1B **Conventional Mission Upgrade Program (CMUP)** is converting the aircraft's nuclear capability to a conventional configuration by integrating a cluster bomb unit. To perform this, 47 new software modules are being designed for conventional bomb-racks to enable switching between 500 pound bombs and 1,000 pound tactical munitions dispensers (TMDs). [SCOTT95] The conversion involves uploading newly designed modules in each weapons bay, modifying the avionics operational flight software, and reloading databases. [AW&ST92] Because software is soft, we are able to produce more flexible solutions like these than any other discipline.

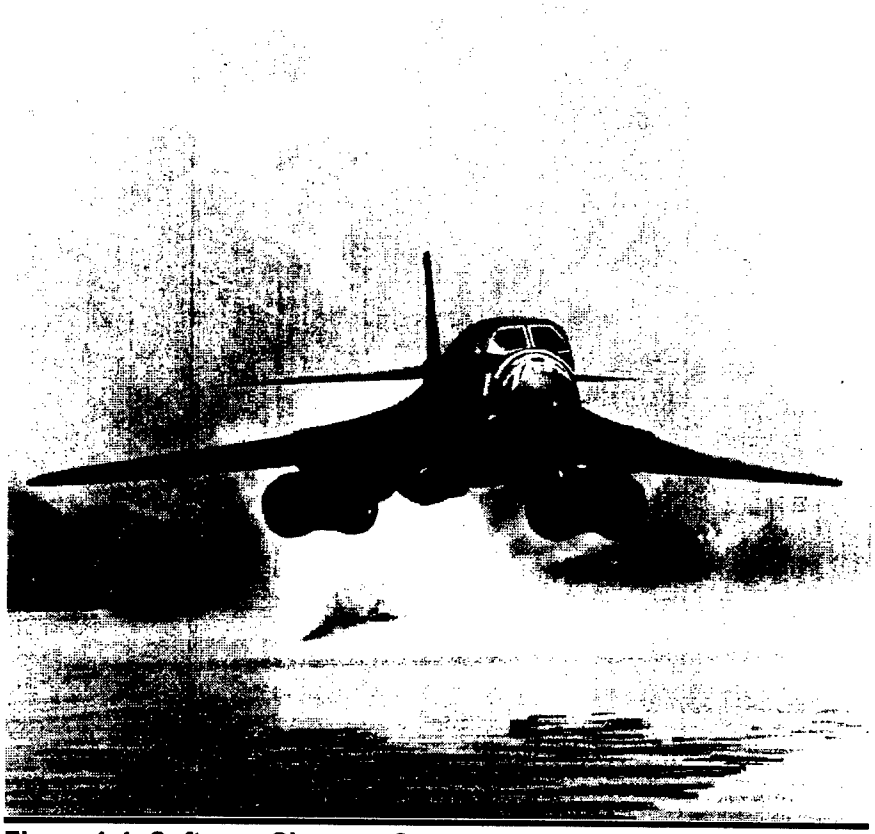


Figure 1-4 Software Changes Convert B1-B from Nuclear to Conventional Capability

That, however, is also why our maintenance costs are more exorbitant than other disciplines. As Glass tells us, we fail again and again to realize that although software is soft, *"it is not so soft that change is free. Far from it, in fact. Change is the biggest money-maker in the software world!"* [GLASS91] We do not realize that unstable requirements are a characteristic of the software beast. We fail to freeze requirements at the outset of the program—when the **Software Requirements Specification (SRS)** is approved. If requirements keep evolving as the software evolves (and especially if there is concurrent hardware development), it is next to impossible to develop

a successful product. Software developers find themselves shooting at a moving target and throwing away design and code faster than they can crank it out.

Poor Problem Solving/Decision-Making

Management is, like all other activities in software development, a problem-solving exercise. It involves deciding what must be accomplished, how to do it, monitoring what is being performed, and evaluating what has occurred. The *"what"* is expressed in the **Software Development Plan (SDP)** *[see Chapter 14, Managing Software Development]* and the *"how"* in an allocation of resources (e.g., schedule and budget). Too often, we stop after these first two steps.

We do not remember that software development is dynamic and our original plans and estimates must be updated. *[Now we are back to the estimation problem above.]* We need more time than projected, new requirements are added, key personnel are sent to other programs. We fail to monitor activities and adjust our plans and resources accordingly. [BLUM92] For example, when change requests are submitted, we fail to make a solid estimate of their impact on our cost and schedule predictions and to change those figures to accommodate the new requirements. We fail to tell our customers (the user or the Government if you are a contractor) that if they want A change badly enough, they will have to pay for it. It is easy enough to understand this problem, but few managers act on it. [GLASS92] We get caught up in trying to please and in the old tail-chasing game of catch up by subscribing to the *"perfect formula for a software disaster!"*

Silver Bullets

Software technology has been the greatest instrument for improving man's efficiency since the Industrial Revolution. When properly used, it provides remarkable competitive advantages. However, while software significantly increases the efficiency of its users, the way software is produced is quite inefficient. Not only is most software hand-crafted — it is produced by manual labor! Where automation has achieved the most significant increases in human productivity, little progress has been made in automating the software process. According to **Capers Jones**, *"The problem is that software has the highest manual labor content of almost any manufactured item in the second half of the 20th Century."* [JONES90]

CHAPTER 1 Software Acquisition Overview

Increasing software productivity and quality is the greatest challenge to our industry. To survive, we must learn to produce software cheaper, better, and faster. In our quest for a more efficient way, we fail to realize there are no easy solutions. We get caught up in the aura of our amazing software solutions to mind-boggling problems in other domains. We become enthralled with wonderful new fads and gadgets that promise to pull us out of our software production drudgeries. We subscribe to the naive belief that a single method or technology (such as computer-aided software engineering (CASE) tools, TQM, or the object-oriented paradigm) will create monumental gains in productivity and quality. We think we can tame the software beast with Silver Bullets when there are none. As **Brooks** tells us,

> *...as we look to the horizon of a decade hence, we see no Silver Bullet. There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.* [BROOKS87]

So how do Silver Bullets cause program failures? Glass tells us *"the search for magic solutions diverts us from the more important search for mundane ones."* We neglect proven, reliable solutions and invest in the hope that a *pie-in-the-sky* magic one will arrive. [GLASS92] Silver bullets are also the reason software technology transfer has been so slow. They make us focus all our attention on one method or technology promising vast improvements, rather than implementing proven ones in parallel. When building large, complex software-intensive systems, we do not realize it takes more than just one tool or technology change for significant process improvement. Multifaceted approaches, including tools, methods, techniques, and processes in parallel, are the proven way to making progress. [JONES94]

In addition, Silver Bullets have rarely been successfully transferred from the laboratory to the production line. The reason they washout is that these technologies have been unable to scale up to the demands of large software-intensive developments. While using new technologies, such as CASE, is a proven method for increasing productivity, we fail in their acquisition and continuous management. We jump on the hype bandwagon and select and acquire them without detailed knowledge of our development processes. Once a financial

commitment has been made, we find the tools do not mesh well with established processes. We do not anticipate the extra time and resources required to train developers to learn the new process required by the tool, or that we may have to bend our old process to fit the tool. Many a program has failed because we have not based our tool selection on a needs-driven process and a pre-acquisition determination that they will, indeed, be beneficial to the people who have to use them. *[See Chapter 10, Software Tools, for a more detailed discussion on software technology.]*

Comparison of Military Software Problems with Commercial Software Problems

Jones claims the military software world lags behind the civilian software world by quite a few years. The factors causing this discrepancy are listed on Table 1-3.

FACTORS WHERE MILITARY SOFTWARE LAGS
It lags in the adoption of functional metrics
It lags in productivity measurement technology
It exceeds all other industries in the production of huge documents
It's schedules are longer than any other kind of software project
It's productivity is lower than for any other industry
It's contracts for software have the highest rates of challenges and litigation
It's contractors rank first in layoffs and downsizing
It's contractors lag in staff benefits and compensation
It's contractors lag in training and education of technical staff
It's contractors lag in training and education of project managers
It's contract software has the highest growth of creeping user requirements
It's contracts associated with SEI maturity levels are much less effective than civilian performance-based contracts

Table 1-3 Where Military Software Lags Behind Commercial Software [JONES95]

Above, you were given a list of the primary reasons why major software-intensive acquisitions fail. Jones has identified why military software programs fail compared to why commercial software programs fail. He says military software acquisition failures out number military successes, especially in logistics support and command and control (C2) applications. In the commercial world, product failure often implies failure or bankruptcy of the company itself, where the industry averages a failure rate that approaches or exceeds **50%**. A general rule of thumb, he explains, is that in any given specific market niche, **10%** of the commercial software products are very successful, **20%** are mildly successful, **40%** are marginal, and **30%** are failures. The factors leading to military software and commercial software failures are listed on Table 1-4.

MILITARY SOFTWARE FAILURE FACTORS	COMMERCIAL SOFTWARE FAILURE FACTORS
Contract is challenged in court	Product fails to achieve market share
Project violates one or more DoD standards	Product loses money
Project adheres to poor civilian practices	Product is readily imitated
Product is very unreliable and of poor quality	Product loses significant litigation
Project fails to meet all requirements	Product generates no ancillary business
Requirements are out of control	Customer support is poor to marginal
Schedules are out of control	User satisfaction is poor to marginal
Costs are out of control	Feature set lags competitors
Project fails critical design review (CDR)	Time to market lags competitors
Product not used or not deployed	Quality levels are poor to marginal

Table 1-4 Military and Commercial Software Failure Factors [JONES95]

SOFTWARE ACQUISITION: The Bright Side

Although software success has been the exception, rather than the rule, there is a bright side to our acquisition story. We are moving forward, we are learning from past mistakes, and we are changing our ways of doing business. Although, we have not advanced to the stage where we can say, "*Beam me up, Scotty*," we are progressing. These changes and extraordinary advances in the production of software have occurred mainly because we have made changes in the way we manage our major software developments. An article published the last week of the Persian Gulf War in *Fortune* magazine stated that:

> *...much of our gee-whiz weaponry has shown that it **does** work, it **is** softening up a fanatic, battle-hard enemy, and it **does** save the lives of Americans and our allies...In Washington the first reaction of those who buy these things for the Government was a mixture of pride and relief. "The good news is that it seems to be working," says John Welch Jr., former Assistant Secretary of the Air Force for Acquisition.* [HUEY91]

Software Success Stories

Our acquisition defeats aside, software has an extraordinary track record and has made an indelible mark on virtually every facet of life in this century. The computer, with software as its brains, landed a man on the moon. Financial transactions are performed at lightning speed while space and aircraft are tracked globally from thousands of eyes and ears above the earth. We can communicate to and from any point on this planet and purchase at the local store enough shrink-wrapped knowledge to prepare and file our taxes or to book our next flight on the Concorde. We waged a war with an arsenal of software-intensive weapons that awestruck the world — friend and foe alike.

> *Just a few months ago, they were the furthest things from our minds, these deadly sleek appliances resting in what General Colin Powell calls his "toolbox" of war implements...America has discovered its arsenal—the damnedest array of stealthy, micro-processed, laser-guided, thermal-imaged, electromagnetically jamming, satellite-vectored weaponry ever imagined.* [HUEY91]

In fact, the military software domain has a few *best in class* attributes, according to Jones, when compared to commercial software, as illustrated on Table 1-5.

FACTORS WHERE MILITARY SOFTWARE EXCELS
It leads in process assessments and process analysis
It leads in numbers of applications larger than I 00,000 function points
It is among the best in reusability research
It is among the best in CASE research
It is the world leader in Ada language research
It is a world leader in configuration control
It is a world leader in requirements traceability
It is among the best in quality control for weapons systems
It has the highest frequency of cost estimating tool usage (although many of these tools are low-end estimating tools not widely used by civilians such as COCOMO and REVIC)

Table 1-5 Where Military Software Excels [JONES95]

The factors to which Jones attributes the success of military and commercial software programs are listed on Table 1-6.

MILITARY SOFTWARE SUCCESS FACTORS	COMMERCIAL SOFTWARE SUCCESS FACTORS
Contract was let without litigation	Product achieves significant market share
Project adheres to relevant DoD standards	Product is profitable
Project adheres to best civilian practices	Product protects unique features
Product is highly reliable with excellent quality	Product prevails in any litigation
Project conforms to all requirements	Product leads to follow-on business
Requirements are stable within 15%	Customer support is good to excellent
Schedules are predictable within 10%	User satisfaction is good to excellent
Costs are predictable within 10%	Feature set is better than competitors
Project passes critical design review (CDR)	Time to market is better than competitors
Product actually deployed and used	Quality levels are good to excellent

Table 1-6 Military and Commercial Software Success Factors [JONES95]

F-22 Advanced Tactical Fighter

Throughout these Guidelines you will find many references to the **F-22 Advanced Tactical Fighter** program, the most software-intensive aircraft ever to be built. In 1992, former Secretary of the Air Force, **Donald B. Rice** explained why the F-22 is a role-model program in testimony before the **House Armed Services Committee**, "*The F-22 program is a fine example of modern management in acquisition...This program is well in hand.*" [RICE92]

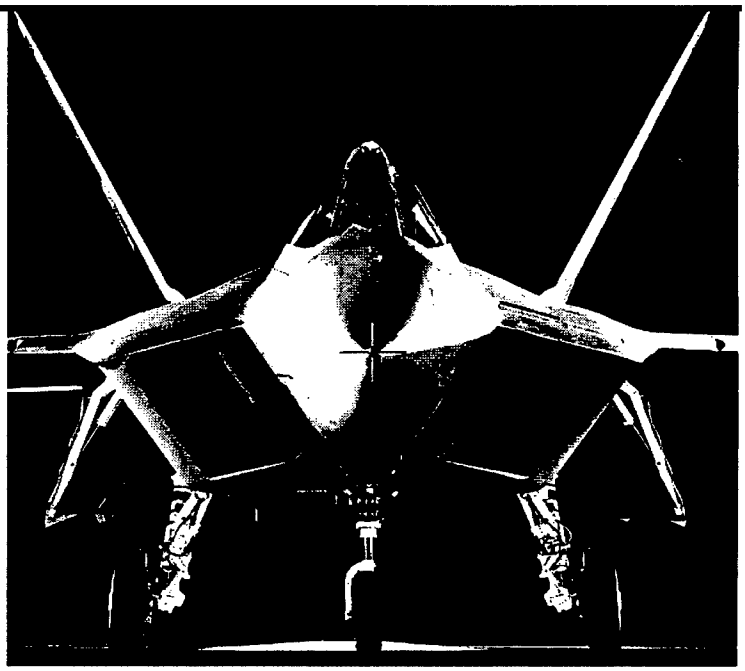


Figure 1-5 F-22 Flagship Acquisition Program

A fully Ada-compliant program, the F-22 is, thus far, an excellent example of ***how to*** acquire and manage a major software-intensive system. Acting on lessons-learned from prior acquisitions, the F-22 procurement strategy was designed to avoid past mistakes by placing special attention on ***software risk***. This strategy includes: (1) early assessment of contractor software development maturity; (2) a commitment to event-driven, rather than schedule-driven milestones; (3) use of integrated product teams to facilitate early identification and correction of problems; (4) use of Ada; (5) use of a common software engineering environment; and (6) preparation of comprehensive **Software Development Plans (SDPs)** that define rigorous quality assurance, risk management, the collection of software cost and quality metrics to track progress, and a commitment by team members to follow and enforce the SDPs. ***The F-22 represents a flagship program on how to achieve software success!***

CHAPTER 1 Software Acquisition Overview

NOTE: The F-22 used the Software Development Capability/ Capacity Review (SDCCR) method *[discussed in Chapter 7, Software Development Maturity]* **developed by ASC/EN, Wright-Patterson AFB, Ohio. The SDCCR was developed for use on embedded and other weapon systems where systems engineering is the predominate management consideration. This method has been updated and is now AFMC Pamphlet 63-103, *Software Development Capability Evaluation (SDCE).***

Major General Robert Raggio, director of the F-22 program office, explains why the F-22 is so important to our national security.

> *If history provides any lessons, chances are that sometime between now and 2012 — the year the F-22 is slated to end production — the US is likely to be in some situation where it needs to gain and maintain air superiority. If the Air Force is going to be called upon to establish that in the future, we must plan for it now. That's exactly what we're doing with the F-22. It combines stealth, super-cruise, integrated avionics, agility, lethality, and supportability to dominate future battles for air supremacy.* [RAGGIO95]

Before a US Senate Appropriations' Defense Subcommittee hearing, **General Ronald R. Fogleman**, US Air Force Chief of Staff, presented the results of numerous analyses and software simulations on the F-22's potential influence in future battles. He said, *"The F-22 gives the joint force commander at least a **30% increase** in his ability to kill the other guy's forces. It preserves, when we look at how it will impact the land battle, **18% less** territory lost."* These software models indicated that four wings (442 aircraft) will reduce:

- Friendly ground causalities by **28%,**
- Armor losses by **15%,** and
- Overall allied air losses by **20%.**

Fogleman also told the Senators that, *"The F-22 is **20 times** more survivable than not only the F-15 as it exists today, but...just about any Western fighter. It's a function of [radar] cross section, supercruise, and integrated avionics."* [FOGLEMAN95[1]] Not only will the F-22 be more survivable than its predecessor, the F-15,

studies show that its life cycle costs will be significantly reduced. As summarized in Figure 1-6, compared to the F-15, the F-22 will:

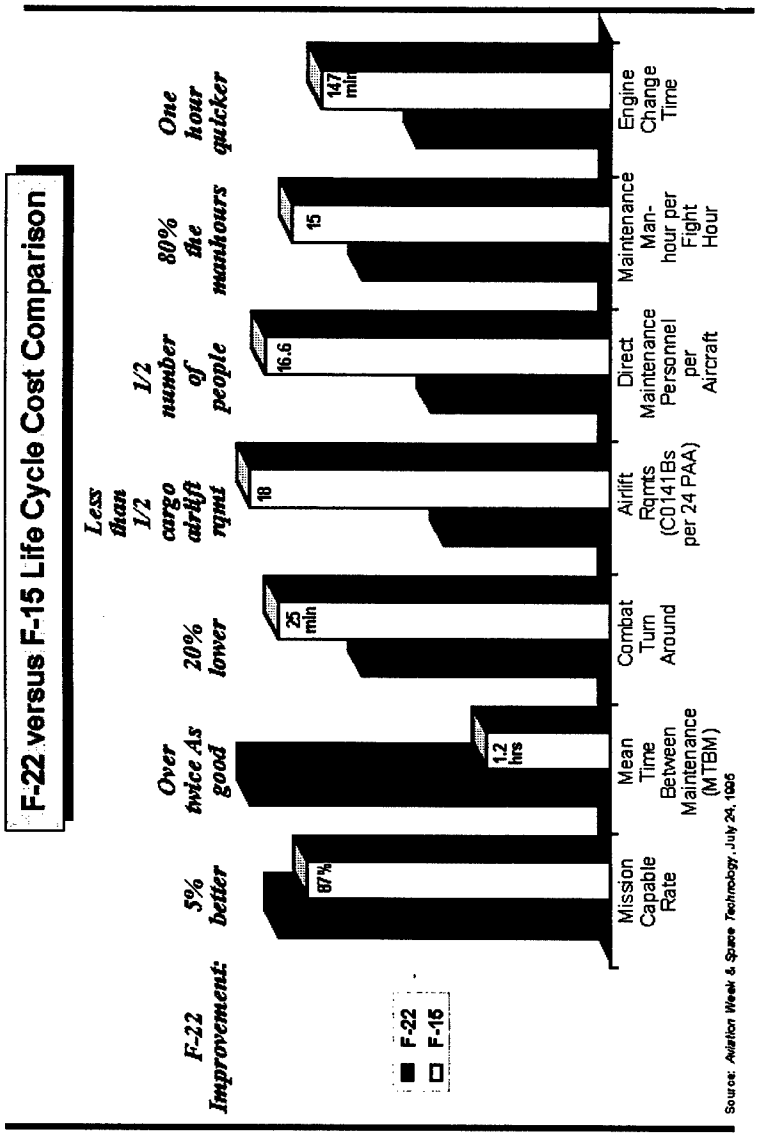


Figure 1-6 F-22 versus F-15 Life Cycle Cost Savings

- Need half the number of maintenance personnel,
- Need half as many C-141B loads to sustain a 24-fighter squadron over a 30-day deployment,

- Cost $500 million less to operate and support over a 20-year period,
- Be able to operate at supercruise, supersonic conditions for extended periods,
- Be equipped with a auxiliary power unit with an 800-hour mean-time-between-failure (MTBF) enabling it to self-start and make air starts at altitudes up to 47,000 feet, at speeds up to Mach 1, and
- Have reduced radar cross section (RCS) with 100% line-of-sight blockage to the engine face. [KANDEBO95]

On February 24, 1995, Fogleman spoke with pride about the F-22 program to the Air Force Association Air Warfare Symposium. He announced that,

> *Let me give you an update on this critical program. Just this last week, the F-22 had a great success story for both the Air Force and the contractor team. The F-22 Air Vehicle Critical Design Review culminated its year-long effort. They reported that the design of the F-22 is* ***mature*** *and that the airplane can be produced and supported.* [FOGLEMAN95[2]]

This means that the F-22's software can be produced and supported. This is an achievement because the F-22 pilot will find himself relying more on software than any previous warfighter. Software will lighten his workload by automating many of those functions a single pilot now performs manually. By displaying less data and more information, the integrated avionics system combines information on enemy aircraft and the total battlefield environment on a single display which the pilot does not have to touch. This system also manages three major sensors: the radar, the electronic warfare system, and the communications/navigation/identification system. The concept behind the software design is to relieve the pilot of operator duties. According to the chief F-22 pilot, Paul Metz, the idea is to *"let the integrated avionics, integrated subsystems, and integrated flight control systems do the mental gymnastics required to operate the aircraft."* [METZ95]

The reason the monumental F-22 software program is on track is the contractor team is using a standard set of Ada tools and proven, defined development processes. According to K. Warren Cannon, Lockheed Martin Corporation's deputy chief software engineer, the success of the software effort is attributable to the following practices:

- All software development teams (comprised of some 20 different companies) are using the same software engineering environment (SEE), the same development processes, and are constantly working to improve those processes;
- An **integrated product development (IPD)** team approach *[discussed in Chapter 4, Engineering Software-Intensive Systems]* helps software designers work with hardware designers for a fully integrated system and for better quality and configuration control;
- All teams have received training on the tool environments and procedures; and
- A **Computer Resources Control Board (CRCB)** exercises strict quality control over all hardware/software systems for the aircraft. [CANNON95]

Following the Dem/Val phase Fogleman discussed above, there was only a 20% growth in software size — modest compared with most software development efforts of this magnitude. According to Cannon, at this point (July 1995) the F-22 software development is only 2% behind schedule and only 5.6% over budget, which is *"almost unheard of"* for a program of this magnitude. [CANNON95] *[See Chapter 10, Software Tools, for a discussion of the Aerospace-Defense/Software Engineering Environment (ASD/SEE) being used on the F-22 program. See Chapter 3, System Life Cycle and Methodologies, for a discussion on the incremental build methodology being used on the F-22.]*

Boeing 777 Transport

The **Boeing** 777 (pronounced "*Triple Seven*") seats 210-420 passengers, has complete *fly-by-wire* cockpit controls, has the most powerful engines ever built for an airliner, and was extensively tested and designed from scratch exclusively by ***software!*** When Boeing, America's leading export manufacturer, undertook the development of this unique program, it was gambling on the aircraft and on itself. [PROCTOR95] With the 777, Boeing undertook the monumental task of re-inventing its manufacturing process and the way it interacts with its customers. Philip Condit, Boeing's President, explained that, *"I firmly believe corporations have life cycles. They grow, they prosper, and if they're not careful, they atrophy and die... We want to be sitting here in 20 years."* [CONDIT95]

Boeing's re-engineering process emphasized ***communication*** and ***teamwork***. The $4 billion ***within cost*** and ***on schedule*** program

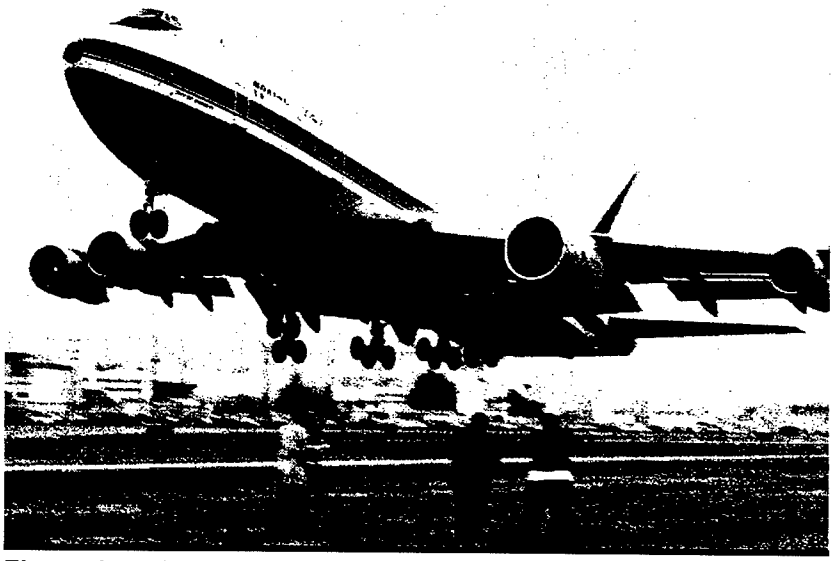


Figure 1-7 Boeing 777 Transport Within Budget and On Time [PROCTOR95]

represents the first time Boeing has included its airline customers and suppliers as members of its design teams. The teams, comprised of people who had previously never worked together, studied successful companies like Ford and Toyota. Their motto was, according to Condit, *"you're not allowed to say, 'Airplanes are different,' because then you don't learn."* [CONDIT94] Boeing also placed heavy emphasis on upfront and continuous risk analysis. According to Kenneth Higgins, Boeing's flight test director, Boeing's top management has provided *"all kinds of cooperation."* [HIGGINS95] A process emphasizing communication and teamwork, the inclusion of customers and suppliers on design teams, and the support from top management all provided an environment where intellectual engagement flourished. The result was teams of people who evaluated design decisions with respect to their impact on future design decisions and system operations — not just trying to solve near-term problems.

Boeing's first *paperless* airplane's entire 3-dimensional design was produced using computer-aided design (CAD) software. The results were the parts fit together much better, ***with 50% fewer design changes and errors***. Software design also eliminated the costly full-scale aircraft mockup used on earlier planes to figure out electrical wire and pipe routings. Rather than steel cables, its fly-by-wire systems which control rudder, elevator, and aileron movement are

completely governed by software. This saves weight, metal fatigue, and allows a more precise flight path. For instance, in a test demonstration, a test pilot rolled the jet inverted and took his hands off the yoke. Warning speakers screamed, ***"TERRAIN! TERRAIN!"*** as the ground raced to meet the plane at lightening speed. Automatically, the 777 completed the roll on its own, leveled its wings to the horizon, and continued on its flight path as if nothing had happened. [COOK94]

777 testing also paved new ground for the airplane company. The 34 flight test pilots each logged 4,500 hours of *ground test* time and 1,800 hours *flight test* time without sitting in an aircraft. The $370-million Integrated Aircraft Simulation Lab (IASL) (nicknamed *Airplane Zero* and the *Skinless 777*) contained three major test facilities. Software systems in one lab tested the function and interoperability of the 777's electrical, avionics, and sensor systems. Software systems in another lab tested the fly-by-wire's electronic flight controls. Another lab tested human factors issues and the interplay among the autopilot, flight director, and aircraft information management system. Higgins exclaimed that the advanced testing performed at the IASL was fantastic, *"We got a head start on problems... we were able to work out [solutions] before hand."* [HIGGINS95] Successful software testing achieved ***a 25% better test rate*** than previous programs. It also helped the 777 program ***maintain schedule.*** [PROCTOR95]

In Addendum C to this chapter *[see note below]*, Robert Lytz explains how the success of the 777 program can be attributed to the fact that Boeing painstakingly addressed those factors that have historically plagued software developments. He cites the *Scientific American* and Brooks' Silver Bullet articles *[see Foreword to this volume and discussions above]* which discuss these common program pitfalls. To address the software size problem, Boeing partitioned the system so software development could occur independently and simultaneously by diverse product teams. Their "*Working Together*" team approach addressed the communications problem; and requirements stability was enforced by a strict change control process. Program management was under control through a software metrics process enacted from program start to end. Schedule and cost were controlled by placing strong emphasis on the integration of COTS into all systems where feasible.

CHAPTER 1 Software Acquisition Overview

NOTE: The Boeing 777 uses the Ada software language. More success stories are found in Chapter 5, *Ada: The Enabling Technology*, where software best practices are enhanced through the use of Ada. Also see Volume 2, Appendix O, Chapter 14 Addendum C, "*On-Board Software for the Boeing 777*," and *CrossTalk*, January 1996, "Software Development for the Boeing 777."

Software Best Practices

In his book, What America Does Right, Robert Waterman explains that by exploring *"in depth, the strategic and organizational reasons why a handful of widely admired American firms do so well...[we] learn from the best; find role models to emulate."* The addendum to this chapter discusses Motorola's strategy for becoming a premier software company. Over the years, Motorola has saved billions of dollars and can now deliver levels of reliability unimagined only a decade ago. Winner of the coveted **Malcolm Baldridge Award**, Motorola shows us that, again, software success is a ***people thing***. Companies who strive to have their employees *become the solution* can dramatically cut development time and costs while increasing product quality and customer satisfaction. [WATERMAN94]

What can we learn from successful software acquisitions? ***Is there a common thread that runs through all that tells us how to proceed and win?*** In studying the winners, there are four basic activities you must perform as a manager that will be stressed and stressed again as you journey through this document:

- You must plan;
- You must manage;
- You must measure, track, and control; and
- You must understand.

Statements made about success in this context, are made in terms that are ***measurable***, ***quantifiable***, and ***repeatable***. Indicators of major software-intensive acquisition success are expressed in terms of cost, schedule, performance, supportability, and most importantly, ***quality*** (as defined and agreed upon by all team members — the customer, maintainer, and developer). Well-managed programs accomplish the following:

- Program baselines (cost, schedule, performance, and supportability) are successfully planned and executed. This includes:
- Identifying risks and employing risk abatement techniques,
- Estimating true costs,
- Estimating realistic schedules, and
- Adequately defining and satisfying performance and support requirements.
- Risks associated with product acquisition are successfully identified, reduced, and managed. This includes:
- Assessing alternative acquisition strategies, and
- Performing acquisition tradeoffs between cost, schedule, and performance.
- Quality and product supportability are accomplished and provided. This includes:
- Employing methods for measuring and implementing improvements associated with product and process modification.

[SYLVESTER91]

SOFTWARE ACQUISITION: Your Management Challenge

In a speech to the 1993 Software Technology Conference, Salt Lake City, Utah, **Mosemann** astounded the audience by saying:

> *It might surprise you, or perhaps even shock you, for me to say that the Pentagon does not want process improvement, it does not want SEI Level 3, or reuse, or Ada, or metrics, or I-CASE, or architectures, or standards. What the Pentagon wants is* ***predictability!*** *Predictable cost, predictable schedule, predictable performance, predictable support and sustainment. In other words, predictable quality.* [MOSEMANN93]

Predictable quality is repeatable quality. Maturing your process cannot be accomplished unless it is first predictable and repeatable. Predictable cost, schedule, performance, and quality is a ***management challenge*** — not a technical one. To advance from an initial, *ad hoc* process to a predictable one you have to institute some basic program controls, the most elementary of which include:

- **Program management.** The fundamental role of a program management system is to ensure effective control of schedule, resources, and quality. This requires adequate planning, clear lines of responsibility, and dedication to a quality product through quality performance. It starts with an understanding of your program's magnitude, complexity, and all other constraints and risks. In the absence of an orderly plan, no estimates can be better than educated guesses.
- **Management oversight**. A disciplined software development program must have senior management oversight, therefore, your contractor should have scheduled in-process reviews. For example, peer inspections should be conducted to assess process compliance, quality of code and documentation, and compliance with requirements. The lack of effective and frequent review of work products typically results in uneven and inadequate process implementation, as well as frequent over (or under) commitments of resources to unrealistic cost and schedules.
- **Quality assurance**. Quality assurance (QA) is a function required for a software development organization to be rated Capability Maturity Model (CMM) Level 2, *Repeatable*. However, quality is built into software through the establishment and use of good software engineering practices (e.g., Cleanroom engineering, peer inspections, etc.). The purpose of QA should be to assure a quality-based process is established and followed to build quality into the software — it is not the purpose of QA to "*inspect in*" quality after the software is built. To be effective, the QA team must have knowledgeable personnel who are an essential part of the development team with an independent reporting line to senior managers. The QA team must also have sufficient resources to monitor the performance of all key planning, implementation, and verification activities. This generally requires a team of about 5% the size of the development team.
- **Change control**. Control of changes in software development is as fundamental to business and financial control as it is to technical control and program stability. To develop quality software on a predictable schedule, user and technical requirements must be established and maintained with reasonable stability throughout the life cycle. Changes that must be made, must be managed and introduced in a systematic, disciplined manner. While occasional requirements changes are common, many of them can be deferred and phased in later. All changes must be controlled or orderly design, implementation, and testing is impossible, and no quality plan can be accomplished. [HUMPHREY89]

SOFTWARE ACQUISITION BOTTOM LINE

The most important thing for a major software-intensive acquisition manager to remember is that, ***"WE ARE BUYING PROCESS AS MUCH AS PRODUCT!"*** The success of your acquisition depends on your developer's ability to deliver a quality software product at a predictable cost in accordance with an established schedule. Without a mature, defined process, the desired product cannot be produced. The ***process*** is critical to program success, and the process that matters is the one in use by your developer. Thus, the bottom line for a successful program is to develop and implement a ***process-driven acquisition strategy***.

A software development organization's ability to produce cost-effective and quality products is based on several controllable factors. These include the development process, the skills and experience of development team, the tools used, product complexity, and environmental characteristics, such as, schedule pressure and communication. [HENDERSON95]

Process-Driven Acquisition Strategy

An important method for addressing and preventing software problems is to treat the entire software development task as ***a process that can be controlled, measured, and improved***. A development process is a set of tasks that, when properly enacted, produces the desired result. An effective software process considers the relationships of all the required tasks, the tools and methods used, and the skill, training, and motivation of the people who enable it. [HUMPHREY89] A **process-driven acquisition strategy** involves the following:

People
Record
Organizational commitment
Capability
Environment
Strategy
Standards

People

Jeffrey Bier, Vice President of Lotus Development Corporation, described the personality traits of creative people in a speech he delivered, "Managing Creatives," at *Industry Week's* annual Managing for Innovation Conference.

> *Creatives are intense. They're always thinking about work. For them, there's no such thing as "Miller Time." They think in their sleep...They are happy to come to work every day and solve puzzles. As one of my people says, "You come in every day and you're given a set of games to play. Fifteen puzzles. Things don't fit and you've got to make them fit." To the creative person, that's heaven!...In general, they work for three things. First, the 'fun' of creation itself. Second, 'admiration' — especially from their peers. Third, the excitement and 'glory' of taking part in a successful creation.* [BEIR95]

Throughout these Guidelines, you will learn that the success or failure of your software acquisition is a *people thing*. It cannot be over stressed that the skills, experience, and creativity of your development personnel are your greatest resource, and the most powerful weapon you have for winning the software battle. ***The most crucial factor in the success of your program is to find the best company with the most talented software professionals who can do exceptional things***. Skills, experience, ingenuity, and a professional commitment

to process improvement and excellence are necessary to ensure your product is world-class with *built-in quality.* Yourdon tells us that the most successful software development *"organizations are focusing their software improvement efforts on the human resource component — often referred to as 'peopleware.'...attention to peopleware issues can literally cause 10-fold productivity improvements, while investments in CASE methodologies, or other technologies rarely cause more than a 30-40% improvement."* [YOURDON92] Remember, *"an idea can turn to dust or magic depending on the talent that rubs against it."* [BERNBACH82]

Record

In a speech to the US Military Academy's graduating class of 1963, General Maxwell D. Taylor, former Army Chief of Staff, explained why experience and a proven track record are so important.

> *Military men who spend their lives in the uniform of their country acquire experience in preparing for war and waging it. No theoretical studies, no intellectual attainments on the part of the layman can be a substitute for the experience of having lived and delivered under the stress of war.*

In Chapter 13, *Contracting for Success*, you will learn that to deliver a successful product, you must select a developer with a ***proven track record*** in developing comparable software of the same application domain, size, complexity, and scope as the software for which you are responsible. As **Parnas** explained, because software is built by trial and error, the only reliable software that can be built is that which has been built before. By choosing a company with a record for success, you will be reducing the risk that your software will be a raw, untried, new product. Experienced developers easily draw from analogies, repertoires, and viable candidates from past, proven successful solutions. There is no substitute for a competent contractor with a record and reputation for having consistently delivered quality software on time, within budget.

Additionally, you must bring together an acquisition staff collectively knowledgeable in software and preferably having a proven track record in acquiring software-intensive systems. Such knowledge depends on a triad of education, training, and experience in software development and/or maintenance. Just sending a few acquisition personnel to a few weeks of software training is no substitute for having personnel with education, training, and actual software experience as part of your staff. If this knowledge is absent, critical acquisition decisions could be made by people who do not fully understand the issues — resulting in costly mistakes.

Organizational commitment

Personnel qualifications are essential, but personnel skills must also be backed with a strong corporate history, a sound software engineering process, and **organizational commitment** to the success of your program. Successful software companies display the knowledge that even the best professionals need a structured, mature, and disciplined environment for them to work as a high performance team. Many companies hire only the best people, but are still plagued with software quality and productivity problems. Software organizations that do not commit to **software engineering discipline** condemn their professionals to endless hours of solving repetitive, technically trivial problems. They may be challenged by the work at hand, but their time is consumed by mountains of uncontrolled detail. Unless these problems are rigorously managed, the best people will not be productive. *Hiring smart people is vital to success, but highly-skilled, experienced people also require the support of a well-managed process to do world-class work.* [HUMPHREY89]

> *There has been a constant struggle on the part of the military element to keep the end—fighting, or readiness to fight — superior to mere administrative considerations...The military man, having to do the fighting, considers that the chief necessity; the administrator equally naturally tends to think the smooth running of the machine the most admirable quality.*
> — Rear Admiral Alfred T. Mahan [MAHAN03]

In a corporate environment, a smooth running mana is the most admirable quality by heightening the readiness to fight your daily software battles. It employee turnover, critical to the successful com software development. For example, the I **Representative Office** (responsible for DoD co observed that a high turnover rate of key software RAH-66 Comanche helicopter program has contril slippages on several critical software componen engine monitoring system, the aircraft systems mar the control database system, and the crewstation inter system. [GAO95[2]]

The commitment to *quality software* is reflected institutionalization of software engineering princip technology, people, training, planning, and the alloca **Quality** evolves from management's pledge to practices. Prevention and early detection of er improvement, and communication with the custo focus of the entire organization. Therefore, it is es developer who is organized to produce quality s management down. The people dedicated to your motivated, mobilized, and trained to accomplish pr Quality software cannot be developed without a plan with a corporate vision and its short-term, medium term goals for success. *Organizational commitme and vital commodity*. Without it, quality softwar

Capability

You will learn in Chapter 7, *Software Developme* a developer's *capability* to deliver quality softwar cost in accordance with an established schedule success of your software acquisition. This can only assessing the maturity of an offeror's establ **development process**. A mature process is one th defined, standardized, structured and constantly i and structure enhance the efficiency and effectiver routine tasks. When developing software, we are stru and managing phenomenally complex logical enti software development involves faultless perform

tasks. A structured, mature process ensures that nothing is forgotten, no problem is overlooked, and that all the pieces fit together. The demands for accuracy, precision, and completeness require formal, orderly methods and procedures. A software organization with a mature process is constantly improving it by recognizing that many routine tasks are often repetitive. They measure, track, and control those tasks to determine how they can be improved and made more efficient. [HUMPHREY89]

Environment

In Chapter 10, *Software Tools*, you will learn that a **software engineering environment (SEE)** is the full set of facilities that support the development process, the purpose of which is to make that process more efficient and accurate. The use of a mature, robust SEE increases productivity and improves product quality by automating all the repetitive activities performed in the production of a software product. Your developer's environment should also include a proven, mature automated process control mechanism. As our software systems grow in size, these environments integrate the support for each separate task into a support framework for the entire process. Because individual software tasks can span multiple development phases, the environment allows these separate activities to be handled in a consistent manner.

Strategy

In Chapter 12, *Strategic Planning*, you will learn that a **development strategy** is your contractor's plan for producing your software. Without a good plan they cannot effectively manage even modestly-sized software developments. Mature organizations also have a strategic planning process that ensures their plans are complete, thoroughly reviewed, and properly approved. As discussed in Chapter 14, *Managing Software Development*, **Software Development Plan (SDP)** defines the work and how it will be performed. The SDP contains a definition for each major task, an estimate of time and resources required, and a framework for management review and control. When properly documented, the SDP can be used as a baseline for actual program performance which can then be compared and tracked. Mature software organizations use SDPs as:

- A basis for getting consensus on the cost and schedule for the program,
- An organizational structure for performing the work,
- A framework for allocating required resources, and
- A record of what was initially committed. [HUMPHREY95]

Standards

As you will learn in Chapter 2, *DoD Software Acquisition Environment*, and Chapter 14, *Managing Software Development*, to ensure your program's success, you must select a contractor who has a defined set of **standards** for excellence in their product and process, and a proven track record for implementing those standards with documented results. A standard is an acknowledged measure of comparison for assessing quantitative or qualitative value and is used to determine the size, content, value, or quality of a product or activity. *Standards bring discipline to the process and measurable quality to the product.*

There are two types of standards used to define the way software is developed and maintained. One class of standards describes the nature of the product to be developed, the other defines the way the work is performed. Typical software product standards pertain to such things as languages, coding conventions, commenting, change flagging, and error reporting. There are also standards for procedures. For instance, there are standards for software reviews and inspections, as well as standard procedures for conducting them. A review standard might define review contents, preparatory materials, participants, responsibilities, and the resulting data and reports. Standard procedures for conducting the review describe how the work is actually to be performed, by whom, when, and the disposition of the results. These standards provide operational definitions about which people can communicate and work toward. [HUMPHREY89]

In the article "Evaluating Software Engineering Standards," *Computer*, September 9, *1994*, a case study of a software development organization was cited that mandated compliance with structured programming standards. Upon investigation, the organization found that only 58% of the software modules they developed complied with those standards. Taking this into account, and realizing that the quality of software documentation can vary significantly from programmer to programmer, it became imperative to establish and

enforce good software engineering and documentation standards. Without these, it is impossible to develop and deliver quality software products. Additionally, meaningful process improvement is only achievable when the organization mandates standards and ensures they are followed. This means you should *choose a contractor who institutes and improves upon formal, repeatable, measurable process controls.* [HENDERSON95]

MANAGEMENT BOTTOM LINE

The **management bottom line** is continuous process improvement *[discussed in Chapter 15, Managing Process Improvement].* This calls for critically evaluating key activities (processes) that span the entire organization. Integrated (cross-functional) product teams (IPT) are used to redesign processes to eliminate unnecessary or nonvalue-added tasks. Process improvement focuses on acquiring (or developing) new technologies to implement the redesigned process. We can improve our processes an order of magnitude when we rely on proven, disciplined methodologies and techniques in our acquisition and development efforts.

Leadership As Well As Management

Creech explains there is a difference between ***managership*** and ***leadership***. Measuring, monitoring, tracking, and assessing are all fundamental managership ingredients which are disciplined methods for gaining and keeping control of your program. But because the success or failure of your software acquisition is a ***people thing***, proactive leadership is essential to get the kind of development processes we want — the performances we need from our team. Leaders set norms and standards for team performance and product quality. Therefore, effective leadership stems from an understanding of how people think, what motivates them, and how to get them to perform at their highest levels.

> *Leadership "of the people thing" means fostering intellectual engagement wherever you find it. There are lots of techniques that have been used in the past to encourage intellectual engagement: design reviews, user challenges, looking at code (by management, not just QA), etc. It is interesting to note that David Cutler, the Windows NT leader in Microsoft, looks at the code.*

> *Where would Chrysler be if Lee Iacocca were never to look at the cars they produce? Yet, software company senior managers seldom take a look at the code.* [HOROWITZ95]

Effective leaders also focus on streamlining the management process. Necessities for successful top management include: (1) a thorough understanding of the organization's customers, their needs, and the environment; (2) recognizing the need for change; (3) defining the strategic business case for change, including a return-on-investment; and (4) focusing on processes instead of functions. [HEIVILIN95] Because process improvement is inherently painful for any organization, proactive leadership is needed to demonstrate a commitment for improvement efforts to succeed.

> *Leadership and command at senior levels is the art of direct and indirect influence and the skill of creating the conditions for sustained organizational success to achieve desired results.* — US Army Field Manual 22-103

Being a Good Leader Means Being a Good Customer

Being a good acquisition manager, thereby being a good customer to your industry developer, means adopting a management philosophy of *cooperation* and *teamwork*. Throughout this century and on into the next, we are going to experience a climate of unprecedented global competition. The margin between winning and losing on the world scene will be very narrow. In the past, there have been adversarial relationships between Government and industry. Many government managers have thought contractors were out to pillage, plunder, and pirate our defense dollars. To win in a global environment, attitudes must change. British Army Major General J.F.C. Fuller, the father of modern armored warfare whose ideas on the use of tanks in combat decisively influenced the Germans and Soviets, defined *"cooperation."*

> ***The Principle of Cooperation***. *Cooperation is a cementing principle; it is closely related to economy of force, and therefore to concentration, but it differs from both of these principles, for while mass is the concentrated strength of the organization and economy of force the dispersed strength which renders the former stable,* ***cooperation*** *may be likened to the muscular tension*

> *which knits all the parts to the whole. Without cooperation an army falls to pieces. In national wars, the value of cooperation is enormously enhanced, fusing as it does, the body and soul of a nation into one intricate self-supporting organism.* [FULLER23]

Enlightenment. Another ingredient for successful management is **knowledge**. Being a good manager means dedicating your time and efforts towards becoming an *enlightened customer*. Learn as much as you can about the concepts, methods, tools, and procedures related to successful software engineering covered in this text. These Guidelines merely present an *overview* of the breadth of knowledge you must gain to truly comprehend the complexities of developing and maintaining major software-intensive systems. Through education you will acquire the fundamental qualities needed to be a good customer: flexibility, common sense, and technical understanding. Flexibility comes from a breadth of experience and openness to new ideas. Common sense and understanding come from the assimilation of information characterized by your ability to compare, discriminate, and judge accurately. *[You can obtain a current listing of available courses either by calling the training points of contact identified in Appendix A or referencing the information on-line through the web addresses given in Appendix B. Also, Appendix F has a list of reading materials to have on your night stand or carry in your briefcase on those long TDYs.]* Remember,

> *To lead, you must know — you may bluff all your men some of the time, but you can't do it all the time. Men will not have confidence in an officer unless he knows his business, and he must know it from the ground up.* [BACH17]

Teamwork: A New Total Force Concept

Industry/military teamwork is how **General Fogleman**, explained we went from a fifth to a first-rate Air Force during World War II. While commander-in-chief of the US Transportation Command, he recalled observing how much of our military transportation comes from civilian industry. He said,

> *I witnessed our civilian airlines, maritime shipping, and the surface transportation industries in action, every day. These industries' use of innovation, new technologies,*

> *and basic ability to track items in their systems were far ahead of the defense establishment. This was the beginning of my awareness that we might do well to...include our partners on the Air Force team. I call this a* ***"new total force concept."*** *Given the political and economic climate in Washington and around the nation, it's absolutely critical that the services adopt such an attitude.* [FOGLEMAN95[3]]

The body and soul of our national assets must be fully exploited so we are prepared to respond to future challenges with the pride and results we delivered in **Operation Desert Storm**. Government and industry have collaborated to solve big problems throughout history, from mobilizing for war to putting a man on the moon. Government recognized the need, industry proposed their solutions, Government reviewed and approved, and industry produced results. Coming together as an integrated unit we can achieve *concentrated-strength-of-organization* and *economy-of-force* with the same passion we have for winning on the battlefield. To win we must forge a partnership by concentrating our strengths, or we will find ourselves overtaken by our more cohesive competitors. As Benjamin Franklin warned at the signing of the Declaration of Independence,

> *We must, indeed, all hang together, or most assuredly we shall all hang separately.* [FRANKLIN76]

Empowerment. Positive things happen when you *empower* team members to pro-actively make incremental and revolutionary changes to their process, thus improving their ability to develop the best product to meet Government needs. This culture is created by allowing active communication and participation to occur. When problems arise, they are *everyone's problems*. A team approach towards solving problems promotes a better opportunity for process improvement, enabling people to do their jobs better and deliver a quality product. This management technique results in disciplined methods, time management, development of process improvement skills, participation and involvement in decision making, boosts in morale, and effective communications. It also results in *economy-of-force*, improved performance, and competitive cost effectiveness. An example of industry empowerment is discussed in Chapter 12, *Planning for Success*. F-22 program managers gave industry the latitude to fly their Dem/Val test programs the way they thought would best demonstrate that program risk had been sufficiently

reduced. Dem/Val was a breeze because only those performance areas the contractors identified as being critical were spot checked. General Patton spoke of empowerment when he said,

> *One of the hardest things that I have to do — is not to interfere with the next echelon of command when the show is going all right.* [PATTON44]

Disciplined trust. **Anthony Salvucci**, former director of engineering and program management, Air Force Electronic Systems Center (ESC), explains that the basis upon which the government/ industry partnership must be built is *"disciplined trust."* Each partner must recognize that they have a different and distinct job, each of which is a necessary element of a successful acquisition process — neither of which is sufficient without the other. The Government has three responsibilities in the acquisition process: (1) to establish requirements, (2) to steward the taxpayers' money, and (3) to be a smart buyer. Industry has the responsibility of interpreting government needs into an acceptable solution, at a reasonable cost, within a reasonable time. Herein lies the distinction between who is delivering the product and who is managing the taxpayers' money. The Government tells industry what we want; the offeror tells the Government how they will deliver it. To be a smart buyer, you must select a winning contractor team that is competent to deliver what they promise and upon whom you can absolutely place your *"disciplined trust."* You must be confident in their ability, because once you award your contract, you have bought their plan — *it's yours, you own it.*

Salvucci explains that if you trust your contractors to deliver as promised, you should not tell them how to manage their process or how to report it. All you should require is access to their management reports (in their format) that let you determine whether, in fact, they have the process they promised and are successfully managing the program. Knowing what is going on, and that the contractor is doing their job by following their plan on schedule, within budget, is all we need to determine whether we are being good buyers and doing our job of stewarding the taxpayers money. [SALVUCCI93]

Team spirit. No one is perfect. Our industry partners are going to make mistakes. When they do, just remember the magnitude of the software task they are undertaking. As Brooks told us, software is more complex than any other construct ever built by man.

[BROOKS87] You must understand there will be glitches, missed schedules, and failure to achieve plans. However, early punishment is not the answer. What is required in every program is an ***enlightened attitude*** to understand why problems occur, to encourage early identification of risks, and to inspire innovative solutions. Consensus on realistic and achievable goals, mutual commitment to meet and/or exceed those goals, and team drive to quickly identify problems and shortcomings will make the difference between a program on a slippery slope to failure and one that plods forward to success.

No team is ultimately successful without a common spirit to hold it together, a shared objective to guide it, and a game plan to help it get there. The key to nonstop process improvement is for you to set aggressive goals and stay actively involved in reviewing your development team's performance towards achieving them. Being a good customer means rewarding success and concentrating resources on those areas needing improvement. As a customer, ***your challenge is to provide the means and leadership for attaining the quality goals agreed upon by all owners of the process.***

REFERENCES

[AW&ST92] "B-1B Displays New Potential in Nonnuclear Tactical Roles," *Aviation Week & Space Technology,* July 27, 1992

[BACH17] Bach, MAJ C.A., "Know Your Men, Know Your Business, Know Yourself," 1917 address to new officers, Robert A. Fitton, ed., Leadership: Quotations from the Military Tradition, Westview Press, Boulder, Colorado, 1990

[BEIR95] Beir, Jeffrey R., "Managing Creatives: Our Creative Workers Will Excel — If We Let Them," speech presented at Industry Week's annual Managing for Innovation Conference, Chicago, Illinois, March 13, 1995

[BERNBACH82] Bernbach, William, as quoted in the *New York Times*, October 6, 1982

[BLUM92] Blum, Bruce I., Software Engineering: A Holistic View, Oxford University Press, New York, 1992

[BOEHM81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981

[BROOKS87] Brooks, Fredrick P., Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987

[BUSWEEK85] "Forget the $400 Hammers: Here's Where the Big Money is Lost," *Business Week*, July 8, 1985

[CANNON95] Cannon, K. Warren, as quoted by David Hughes, "F-22 to Counter 21st Century Threats; Avionics Automates Many Cockpit Functions," *Aviation Week & Space Technology*, July 24, 1995

[CONAHAN95] Conahan, Frank C., "Defense Programs and Spending: Need for Reforms," Testimony Before the Committee on the Budget, House of Representatives, GAO/T-NSAID-95-149, April 27, 1995

[CONAN-DOYLE91] Conan-Doyle, Sir Arthur, "The Five Orange Pips," The Adventures of Sherlock Holmes, 1891

[CONDIT94] Condit, Philip, as quoted by William J. Cook, "The End of the Plain Plane: Boeing Co.'s Boeing 777", *US News & World Report*, April 11, 1994

[CREECH94] Creech, Gen Bill, The Five Pillars of TQM: How to Make Total Quality Management Work for You, Truman Talley Books, Button, New York, 1994

[DeMARCO87] DeMarco, Tom and Timothy Lister, Peopleware: Productive Projects and Teams, Dorset House Publishing Co., New York, New York, 1987

[DiNITTO92] DiNitto, Samuel, A., Jr., "Rome Laboratory," *CrossTalk*, Software Technology Support Center, June/July 1992

[DSB87] Office of the Under Secretary of Defense for Acquisition, *Report of the Defense Science Board Task Force on Military Software*, September 1987

[DSB94] Office of the Under Secretary of Defense for Acquisition & Technology, *Report of the Defense Science Board Task Force on Acquiring Defense Software Commercially*, June 1994

[FOGLEMAN95[1]] Fogleman, Gen Ronald R., as quoted by John D. Morrocco, "Balancing Act in Congress; Big F-22 Budget Drives Search for Flaws," *Aviation Week & Space Technology*, April 10, 1995

[FOGLEMAN95[2]] Fogleman, Gen Ronald R., "Getting the Air Force Into the 21st Century: The Ability to Model and Simulate Combat," speech presented to the Air Force Association Air Warfare Symposium, Orlando, Florida, February 24, 1995

[FOGLEMAN95[3]] Fogleman, Gen Ronald R., "Aerospace Industry Has Earned 'Full Partner' Status," *Armed Forces Journal International*, June 1995

[FRANKLIN76] Franklin, Benjamin, statement at the signing of the Declaration of Independence, July 4, 1776, *The Whistler*, 1779

[FULLER23] Fuller, MGEN J.F.C., The Reformation of War, Hutchison & Co., London, 1923

[GAO79] General Accounting Office, *Contracting for Computer Software Development—Serious Problems Require Management Attention to Avoid Wasting Additional Millions*, FGMSD-80-4, November 9, 1979

[GAO95[1]] General Accounting Office, *Tactical Aircraft: Concurrency in Development and Production of the F-22 Aircraft Should Be Reduced*, GAO/NSIAD-95-59, April 1995

[GAO95[2]] General Accounting Office, *Comanche Helicopter: Testing Needs to be Completed Prior to Production Decisions*, GAO/NSIAD-95-112, May 1995

[GIBBS94] Gibbs, W. Wayt, "Software's Chronic Crisis," *Scientific American*, September 1994

[GLASS91] Glass, Robert L., Software Conflict: Essays on the Art and Science of Software Engineering, Yourdon Press, Englewood Cliffs, New Jersey, 1991

[GLASS92] Glass, Robert L., Building Quality Software, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992

[GREENE90] Greene, Col Joseph, Jr., as quoted by Evelyn Richards, "Pentagon Finds High-Tech Projects Hard to Manage: The Army Still Awaits Computerized Battlefield," *The Washington Post*, December 11, 1990

[HEIVILIN95] Heivilin, Donna M., testimony before the Subcommittee on Readiness, Committee on National Security, House of Representatives, *Defense Infrastructure: Enhancing Performance Through Better Business Practices*, GAO/T-NSIAD/AIMD-95-126, General Accounting Office, March 23, 1995

[HENDERSON95] Henderson, COL Jerry M., "Will Army Software Win the Information War?" *Army RD&A*, July-August 1995

[HIGGINS95] Higgins, Kenneth, as quoted by Paul Proctor, "Early Modeling Helps Speed 777 Flight Testing," *Aviation Week & Space Technology*, June 12, 1995

[HOROWITZ95] Horowitz, Dr. Barry M., personal correspondence with Llyod K. Mosemann, II, September 8, 1995

[HUEY91] Huey, John and Nancy J. Perry, "The Future of Arms," *Fortune*, February 25, 1991

[HUMPHREY89] Humphrey, Watts S., Managing the Software Process, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989

[HUMPHREY95] Humphrey, Watts S., A Discipline for Software Engineering, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995

[JONES90] Jones, Capers, as quoted by Evelyn Richards, "Society's Demands Push Software to Upper Limits: More Computer Crises Likely," *The Washington Post*, December 9, 1990

[JONES94] Jones, Capers, Assessment and Control of Software Risks, Yourdon Press, Englewood Cliffs, New Jersey, 1994

[KANDEBO95] Kandebo, Stanley W., and David Hughes, "F-22 to Counter 21st Century Threats," Aviation Week & Space Technology, July 24, 1995

[KEENE91] Keene, Charles A., white paper "Lessons-Learned: Nuclear Mission Planning and Production System," AF Strategic Communications-Computer Center (SAC), Offutt AFB, Nebraska, January 17, 1991

[MAHAN03] Mahan, RADM Alfred T., Naval Administration and Warfare, Little, Brown, & Co., Boston, Massachusetts, 1903

[MARCINIAK90] Marciniak, John J., and Reifer, Donald J., Software Acquisition Management: Managing the Acquisition of Custom Software Systems, John Wiley & Sons, Inc., New York, 1990

[METZ95] Metz, Paul, as quoted by David Hughes, "F-22 to Counter 21st Century Threats; Avionics Automates Many Cockpit Functions," *Aviation Week & Space Technology*, July 24, 1995

[MOSEMANN93] Mosemann, Lloyd K., II, as quoted in *Ada Information Clearinghouse Newsletter*, Vol XI, No. 2, August 1993

[PARNAS85] Parnas, David Lorge, "Software Aspects of Strategic Defense Systems," *American Scientist*, September-October 1985

[PAT92] Air Force Systems Command, *Software Process Action Team Final Report: Process Improvement for Systems/Software Acquisition*, June 30, 1992

[PATTON44] Patton, GEN George S., Jr., speech to the Third Army, "A General Talks to His Army," June 1944

[PAULSON79] Paulson, Paul J., as quoted in the *New York Times*, May 4, 1979

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Second Edition, McGraw-Hill, New York, New York, 1992

[PROCTOR95] Proctor, Paul, "Early Modeling Helps Speed 777 Flight Testing," *Aviation Week & Space Technology*, June 12, 1995

[RAGGIO95] Maj Gen Robert, as quoted by Stanley W. Knandebo and David Hughes, "F-22 to Counter 21st Century Threats," *Aviation Week & Space Technology*, July 24, 1995

[RICE92] Rice, Secretary Donald B., "Reshaping for the Future," speech delivered to the House Armed Services Committee, Washington, D.C., February 20, 1992

[RICHARDS90[1]] Richards, Evelyn, "Society's Demands Push Software to Upper Limits: More Computer Crises Likely," *The Washington Post*, December 9, 1990

[RICHARDS90[2]] Richards, Evelyn, "Pentagon Finds High-Tech Projects Hard to Manage: The Army Still Awaits Computerized Battlefield," *The Washington Post*, December 11, 1990

[SALVUCCI93] Salvucci, Anthony, "Vision for a New Acquisition Process," speech presented to the Armed Forces Communications and Electronics Association (AFCEA) luncheon, November 19, 1993

[SCOTT95] Scott, William B., "Weapons, Avionics Upgrades Expand B-1B Options," *Aviation Week and Space Technology*, February 13, 1995

[SYLVESTER91] Sylvester, R., "Process Improvement for Systems/Software Acquisition," briefing presented to Rome Laboratories, September 23, 1991

[VESSEY84] Vessey, GEN John W., as quoted in the *New York Times*, February 25, 1984

[WATERMAN94] Waterman, Robert H., Jr., What America Does Right: Learning from Companies That Put People First, W.W. Norton & Company, New York, 1994

[WHITTEN95] Whitten, Neal, Managing Software Development Projects, John Wiley & Sons, Inc., New York, New York, 1995

[WOLFE27] Wolfe, MGEN Sir James, "Reflections on the Battle of Culloden," as quoted by Liddell Hart, Great Captains Unveiled, 1927

[WULF90] Wulf, William, as quoted by Evelyn Richards, "Society's Demands Push Software to Upper Limits: More Computer Crises Likely," *The Washington Post*, December 9, 1990

[YOURDON92] Yourdon, Edward, Decline & Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992

CHAPTER 1
Addendum

Software Solution

Editor's Note

Effective management of software-intensive development activities requires total commitment from the top to the bottom of an organization. This article, reprinted with permission from the Software Quality Institute University of Texas at Austin, illustrates how one company recognized that software technology represents a major technological revolution, mobilized its executives to understand and address the challenge, and has developed its ***software solution****. After you have finished these Guidelines, develop* ***your*** *software solution.*

SOFTWARE SOLUTION: Motorola's Strategy for Becoming The Premier Software Company

Software is growing at a revolutionary pace. To continue our corporate success, Motorola must be equipped to compete in this world of drastic technological change. Our response to this industry upheaval is a systematic change in our software culture that meets the urgent needs of the future. When we have completed this process, we will not only be a competitor in the business of the future, we will continue to be an industry leader. We call this long-term initiative the ***Software Solution.***

Call to Action

In January 1991, Motorola gathered a team of 27 officers and facilitators for a three-day training session on software. Chairman and CEO George Fisher asked each participant to accept the challenge

of changing Motorola's software culture. Known as the Senior Executive Program (SEP) on Software, this team is charged with personal commitment and involvement to start Motorola on the path to preeminence in software. The team is dedicated to making software a core competency within the company by 1998.

Driven by Motorola's senior executives, with CEO sponsorship, the Software Solution is a cooperative effort between SEP, the Software Engineering and Technology Steering Committee, Motorola University, and the Motorola Software Research and Development. Now in their third year, the executives are divided into smaller teams in these focus areas: Vision/Marketing, Management, Process/ Metrics, Tools/Technology, People, and Benchmarking.

In early 1991, I charged a team of Motorola's senior executives with the responsibility of assessing the importance of software to Motorola's future. I felt we could generate significant new business opportunities with improved software development performance. The SEP group conducted its own analysis and came to the same conclusion. This group has taken leadership responsibility. I continue to be an active participant. I ask all Motorola executives and managers to seek the information and training needed to understand the software issues in their organization and contribute to the software solution. We need their commitment and personal involvement.
— George Fisher, *Chairman and Chief Executive Officer*

Change Inside for Change Outside

The mission of the Software Solution is to create, initiate, and execute software initiatives that cause change to occur at an accelerated rate — in effect, to create the software solution. For Motorolans, this means developing the approaches that will give us the capabilities we need to achieve world-class status in software. We believe this competence is fundamental to achieving our business goals. The strategies we are pursuing urgently demand that we achieve excellence in developing and using software. The Software Solution will be a pivotal factor in achieving our growth and financial goals in the next decade.

CHAPTER 1 Addendum

Radical Surge in Software Growth

The growth in software-driven products dominates all facets of business, including management information systems (MIS), financial reporting systems, manufacturing systems, product design, customer order systems, and consumer products. As Motorola continuously improves its software development processes, it will simultaneously improve quality, reduce cycle time, increase productivity, and increase customer satisfaction in all areas of operations for both internal and external customers. Increasingly, software — not hardware — is playing a role in the functionality of electronic products.

To Lead in Systems, We Must Lead in Software

Globally, the software application industry is estimated to be about $175 billion and it is growing at a double-digit rate. While you might say "*software isn't my business,*" in truth, it is everyone's business. Software competence is the key to Motorola's continued leadership in systems. Because product differentiation depends less on hardware and more on software, hardware prices and profit margins are decreasing. Software allows Motorola to provide the customer with value-added services and features and to earn a higher profit margin. We must learn to leverage the strategic advantage of software. For example, here are the opportunities when we sell a flexible platform radio system:

- Software enables us to differentiate our hardware from that of our competitors;
- Features can be tailored for customers' needs;
- Software can provide for future needs, ensuring a continuing customer relationship; and
- Customer demand for internally produced software applications may spawn a new service business.

Motorola is not alone, however, in its quest for software excellence. Dozens of our competitors are pursuing similar initiatives; among them are Hewlett-Packard, IBM, Toshiba, Philips, and many companies in Europe and Japan.

People Make It Happen

While tools and process improvements are important in our quest for the Software Solution, people are our most important resource. This is especially true in software engineering, which is a people-intensive endeavor. In fact, there are approximately 12,000 software engineers and others involved in software functions currently employed at Motorola.

To achieve our 1998 goal, we will need to attract, retain, and develop world-class software talent and provide a culture that permits such talented people to achieve excellence. To create a software-friendly environment, technical and nontechnical managers alike will need a general knowledge of software and its development process. This will also help managers to seek people with intellectual engagement in the design. Such people think beyond the immediate design issues, and think across disciplines, e.g., software logic designers would think about both software development and future user needs, continually assessing the impact of their design decisions on those future needs. Intellectually engaged people anticipate and are disciplined. Anticipation is the key to creativity, with discipline being the key to realized creativity — a successful program has both.

Changing the Way We Work and Our Workplaces

Writing software is a team effort, and, as Motorola continues to move toward systems development, these teams will grow in number and size. At the same time, software and hardware requirements must be integrated, and this requires the smooth interaction of many teams.

The nature of our work will require advanced skills in systems design, software architectures, and — most of all — software program management. A commitment to training is absolutely essential. Even the physical work spaces of our engineering people will change to accommodate the unique requirements of software design work and teamed programs.

CHAPTER 1 Addendum

Trai ıing Leads the Way to Improvement

To effectively manage in the new software culture, managers need to understand the complexity and time requirements of software engineering. To help them gain this knowledge and enable them to set realistic goals for their businesses, courses are being developed by Motorola University's College of Software Engineering and Technology. Through this training, they will gain the skills and tools to expertly manage the members and teams and bring a program to a successful conclusion, on time and within budget.

The strategies we are pursuing demand that we achieve excellence.

In all areas of Motorola, software engineering will increase as part of an engineer's job, and basic software development ability will be a requirement in all engineering disciplines. In addition, on-going training and education are paramount, as new software development tools and techniques are introduced in this rapidly changing field. By gaining new skills, our engineers can improve the software development process and reduce the engineering time to market.

Becoming a Part of the Software Solution

Clearly, the Software Solution will require effort from all. Where does one begin?

- Get the training to understand the issues and the tools to address them.
- Make software issues a featured item in your planning processes for business, strategy, technology road maps, and organizational and management development plans.
- Learn as much as you can about software development from industry leaders, or confirm that your current software objectives correspond to those of the best in class.

Software is everyone's business.

Understanding the Software Revolution

Along with Motorola's courses and programs, the following publications will help you build an awareness of software issues and create software solutions in your business.

Fred Brooks, The Mythical Man-Month, Addison Wesley, 1975. This classic text, written by a manager of one of IBM's largest software development efforts depicts many of the problems (and some of the solutions) associated with software development. Written in an informal style, the book contains many anecdotes.

Fred Brooks, "No Silver Bullets," *IEEE Computer*, 1987. In the style of his other works, Brooks considers recent issues such as object-oriented methodology and Ada. He concludes that improvements are occurring but there is no Silver Bullet, or simple solution, for dealing with software development.

Tom DeMarco, Controlling Software Projects, Yourdon Press, 1982. This is a thorough and pragmatic treatment of many aspects of software development management, including program metrics and software quality issues.

Tom DeMarco/Tom Lister, Peopleware: Productive Projects and Teams, Dorset House, 1987. This is a discussion of the social, ergonomic, and workplace factors that influence software development and management.

Richard Fairley, Software Engineering Concepts, McGraw Hill, 1985. Fairley's is one of the first text books to address fundamental issues in software engineering development and management.

Ann Miller, "Engineering Quality Software," *Motorola Six Sigma Series*, Addison-Wesley, 1992. This paper discusses software defect detection and prevention techniques, with emphasis on a new technique called preview. Preview has been successfully used in Motorola GSTG and Satellite Communications.

Richard Thayer, Software Engineering Project Management, IEEE Computer Society Press, 1988. This tutorial contains numerous short articles on a wide range of topics related to software engineering management.

Edward Yourdon, The Decline and Fall of the American Programmer, Simon & Schuster, 1992. This book anticipates that America will lose market share if businesses do not improve their effectiveness in managing software development processes and provides guidance on how to do so.

CHAPTER 1 Addendum

We know it represents a massive change in culture for Motorola to achieve the Software Solution. It is also an absolute business imperative that we achieve this change in a most urgent fashion. Culture change is a learning process. And at first, the process is frustratingly slow. To achieve cultural change, you need leadership, vision, strategy, timelines, training, communication, goals and measurements, baselining, support structures, and — most of all — a sense of urgency.

Clearly we have a very long road to travel before we can claim significant progress across the board, but islands of excellence are beginning to appear.

— Bill Millon, *Vice President and Director Software Solution and Technology.*

Blank page.

CHAPTER

2

DoD Software Acquisition Environment

CHAPTER OVERVIEW

The world witnessed the awesome force of America's Information Age weaponry during ***Operation Desert Storm****. Smart weapon systems defended thousands of allies and saved as many lives. Stealth aircraft, armed with smart munitions, made the surgical strike a dinner table buzz word. Command and control systems orchestrated the largest, fastest deployment of military might and sensational battlefield tactics the world has seen. All these systems shared a common, critical benefactor:* ***software****. The significance of software to DoD is incalculable.*

With shrinking defense dollars as incentive and Operation Desert Storm as a baseline, DoD has modernized its concept of the battlefield and its plans to fight and win through increased reliance on software-intensive systems. These systems provide the flexibility to adapt to changing threats and amplify force strength with the versatility and leverage needed to compete and win. This places unprecedented demands on the acquisition community to equip the warfighter with dependable, maintainable, lethally accurate, software-intensive systems that are affordable and delivered on time. As you learned in Chapter 1, Software Acquisition Overview, we have not always been successful in achieving this goal when it comes to software. Changes needed in the way we procure these systems are reflected in DoD acquisition reforms.

In today's Information Age, the commercial sector is the leader in advanced technologies. With post-Cold War defense spending declining, there is a critical national need to merge the defense and commercial industrial bases. In this chapter you will learn that DoD has made a commitment to ***doing business more like business*** *by implementing commercial purchasing practices. Defense-unique specifications and standards are not to be applied to future DoD acquisitions. However, software poses an extra challenge. The industry is still relatively immature as reflected in the high risk of producing quality software, on time, and within cost. Therefore, where software is concerned, some waivers to the new acquisition reform mandates are to be implemented to minimize these risks and ensure our suppliers are successful in delivering world-class software-intensive systems to the warrior.*

Blank page.

CHAPTER

2

DoD Software Acquisition Environment

THE INFORMATION AGE HAS DAWNED

Software-intensive systems have forever changed the American military's concept of the battlefield. **General Colin L. Powell,** former Chairman of the Joint Chiefs of Staff, wrote about his "*toolbox*" of software technology in an article for *Byte* magazine the year after the Gulf War.

> *The **Information Age** has dawned in the armed forces of the US The sight of a soldier going to war with a rifle in one hand and a laptop computer in the other would have been shocking only a few years ago. Yet, that is exactly what was seen in the sands of Saudi Arabia in 1990 and 1991.* [POWELL92]

More important than the military hardware upon which it depends, our constantly changing arsenal of software distinguishes us from every other advanced military on the globe. The character, disposition, capability, usability, interoperability, maintainability, and flexibility of these software-intensive systems gives us the technological edge to compete and win in an ever-changing volatile world environment. You, as the next generation of managers, are entering an environment where DoD's appetite for software is insatiable. Why? Because software accomplishes the following:

- It provides the flexibility to adjust to previously unknown threats,
- It allows us to do more with less,
- It increases the capabilities of airmen, soldiers, sailors, engineers, managers, and battlefield commanders alike, and
- It provides the versatility and leverage we need to compete and win. [PETERSEN92]

We all remember **F/A-18 Hornets** blasting off carrier decks, **AH-64 Apaches** thundering over the desert, **M1A1 Abrams** tanks rumbling over the sands, Cruise missiles homing in on targets, and **Patriot missiles** intercepting incoming enemy SCUDs. All these pieces of hardware became overnight sensations. But the real hero — the one that processed, analyzed, guided, distributed, and gave these systems their prowess — was the invisible, most powerful weapon we possess — ***our software!*** [TOFFLER93]

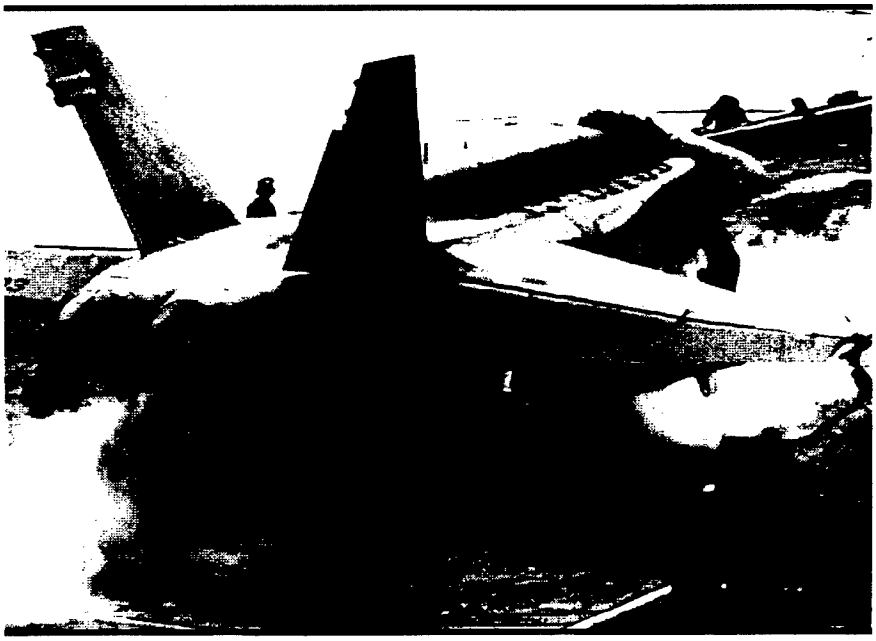


Figure 2-1 F/A-18 Hornet Blasting Off Deck of USS Saratoga

In a speech on the role of software in modern warfare, **Lieutenant General Robert H. Ludwig**, former HQ USAF/SC, explained that, "*In Desert Storm men and machines went off to war with something the world has never seen...software.*" When modern weapons systems are referred to as being "***smart***," it is because software

provides their brains. For instance, by retrofitting them with smart software-intensive components, even the I.Q.s of stupid bombs can be raised. As Ludwig succinctly stated, the "*Fly-by-wire F-16C...without software,*" is nothing more than, *"...a 15-million dollar lawn dart!"* [LUDWIG92]

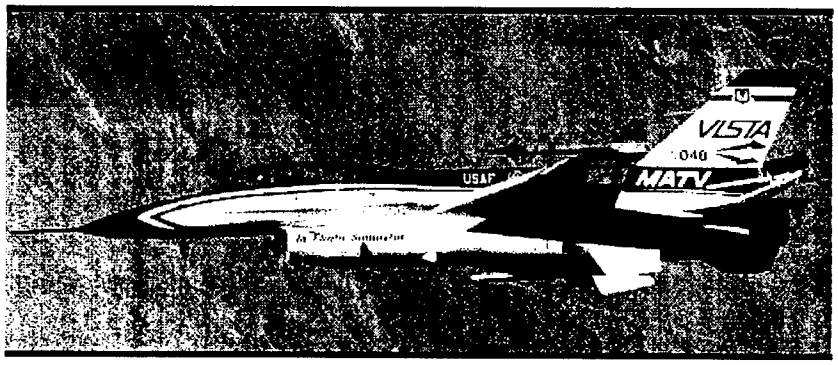


Figure 2-2 Without Software the F-16 Is Only a 15-Million Dollar Lawn Dart

The acquisition and management of software-intensive systems (hardware, firmware, software, documentation, and people) is a relatively new field of endeavor. As recently as the Vietnam War, the **F-4 Phantom** used virtually no software in its weapon systems and software was used sparingly in other defense applications. Back then software-intensive systems were characterized by big workhorse main frames, occupying large rooms, using thousands of watts of electricity, tons of air conditioning, punched card inputs, with long overnight turnarounds. During the 1970s, the rapid evolution of sophisticated electronic circuitry gave us smaller processors producing more computing power for a fraction of the cost. These advances, compounded by more demanding requirements, dramatically increased DoD's software use. Figure 2-3 (below) represents a summary of Air Force and NASA software-intensive systems growth trends, illustrating a progressive increase in software systems size.

Today, software accomplishes many functions formerly performed by specialized hardware, and in most cases, formerly impossible by hardware alone. An example is the **B-2 bomber**. To cut down on its radar profile (or cross-section), it has no vertical surfaces; e.g., it has no tail. Software controls all the aircraft's directional stability. Another example is the automated flight controls on the **F-117 stealth fighter**. More than any other system component, software makes stealth possible. [DANE90]

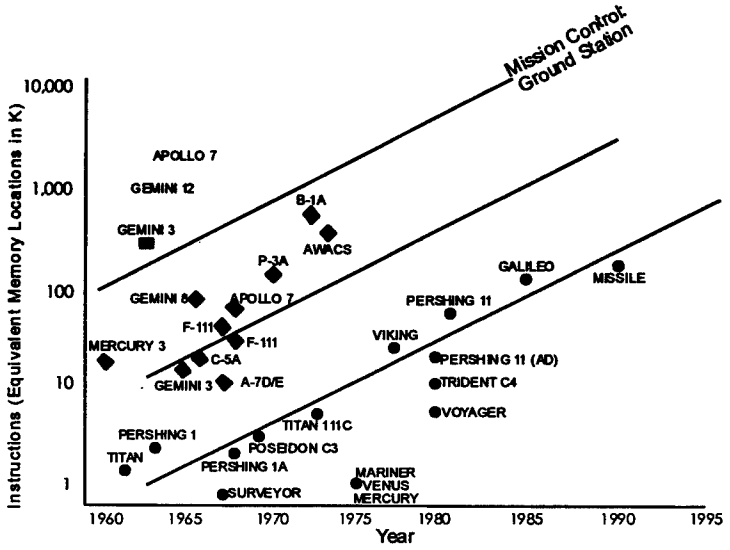


Figure 2-3 Software-Intensive Systems Growth

Software's significance continues to grow. Take, for example, the **F-22** which is scheduled for initial production in 1997, with full production deliveries after the year 2000. [SMITH94] 80% of its functionality is dependent on software which comprises 30% of its engineering and manufacturing development (EMD) cost. Software helped design it, is helping to build it, and will help fly it. **Lieutenant General Jim Fain** (USAF retired), while F-22 Program Director, described software's importance when he said, *"The only thing you can do with an F-22 that does not require software is to take a picture of it"* [and today even the camera has software!] [FAIN92]

Software not only helps us fight and win, it enables us to train and wargame. Simulators used to train and models used by strategists, are enabled by software and sensors. As **Admiral James B. Busey, IV**, (USN retired) claims, simulators relying on **artificial intelligence (AI)** software provide high density, fast, effective, and inexpensive ways for us to prepare the warfighter for possible far-flung encounters and unforeseen conflicts. He explains that in future wars, there will be too much information, too widely spread, for any one individual or single unit to cope without help from *intelligent* software systems. He says that,

CHAPTER 2 DoD Software Acquisition Environment

> *The Defense Department's science and technology strategy places strong emphasis on synthetic environments using computing science for distributed interactive simulation. Among computer sciences developments are automation and robotics; aided or automatic target recognition; and distributed command, control, and communications. The fundamental concept of a machine that can process artificially sensed information, make optimal decisions based on this information and on well-defined objectives, and translate those decisions into actions is a guiding and unifying theme for research in all major aspects of this field.* [BUSEY95]

Where databases merely store information, AI systems ***use*** information. They treat data as knowledge — not just surface patterns, but meaningful information that has consequences, that makes things happen. [HAYES93] DoD uses AI models and simulators during concept exploration for new or upgraded weapon systems acquisitions to expand and evaluate the range of technical, operational, and system alternatives. They are also used for test and evaluation exercises and for planning and decision aids to stretch the ability of commanders to train, plan, and employ their forces. [BUSEY95] For example, the projected fifth Navy **Seawolf** (a smaller, less expensive version than its predecessors) went on a test cruise through cyberspace in mid-May 1995. Using simulation design software and a developed-in-house animation package, seven design/build teams put their vision of the future stealth submarine through its paces. Engineers were able to assemble a software mockup of the Seawolf and analyze its anticipated performance characteristics in a virtual undersea environment. Through their software models, design teams took their Navy customers on a cyberspace tour of the futuristic *fly-by-wire* vessel. Similar to the design of the **Boeing 777** *[see Chapter 1, Software Acquisition Overview]*, software is enabling shock-level tests (anticipated effects of different types of impact damage) to be run on various Seawolf components to determine where *ruggedized*, versus *militarized*, equipment can be used. [ROOS95]

July 11, 1995 marked another successful example of how software saves time and test resources. On that date the durability testing of the **C-17** airframe was completed under the full-scale engineering and manufacturing development (EMD) phase of the program. 60,000 simulated flight hours were logged — the equivalent of two

design lifetimes — of which more than 17,000 simulated flights were conducted — the equivalent of a 60-year operational life. Airframe loads simulating 25 different mission profiles, ranging from airdrops to short-field landings, were enacted by more than 260 software-intensive hydraulic actuators. Movement data were processed and analyzed from over 1,000 strain gauges and deflection monitors. Approximately 11% of the flight profiles were performed in the high stress environment of flight below 2,000 feet at speeds above 300 knots. Several weeks ahead of schedule, EMD testing requirements for the C-17 detailed airframe specification were satisfied without leaving the ground. [SMITH94]

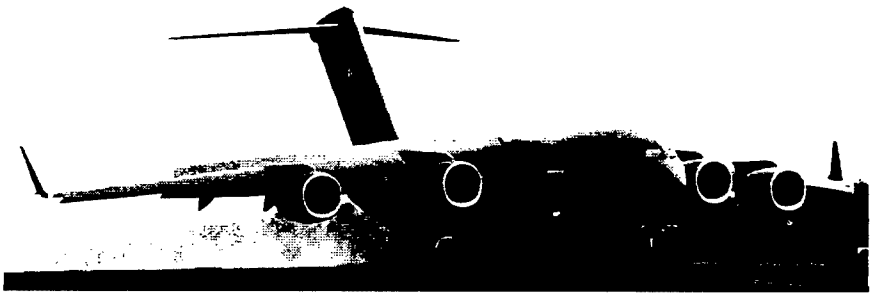


Figure 2-4 C-17 Aces EMD Phase with Simulated Durability Testing

DoD SOFTWARE DOMAINS

To fully understand the impact of software on the DoD acquisition environment and the crucial role and critical demands of software management, you must have a basic grasp of DoD software domains and changing trends in the DoD acquisition process. So, just what do we mean by "***software***?" It is through the medium of software that humans are able to tell the computer's electronic circuitry how we want it to behave. However, a computer does not understand the language we speak as humans; it only understands electronic signals representing the **binary numbers "0"** and "**1**." The computer interprets groups of binary numbers as instructions (or data) and performs operations according to well-defined **instruction sets**. ***Software applications, therefore, are sets of related instructions performed to fulfill specific user needs***. Software, then, can be defined as

> *...the totality of applications usable on a particular kind of computer, together with the documentation associated with a computer or application, such as manuals, diagrams, and operating instructions.* [PARKER89]

Because software applications and data are *intangible* (they cannot be seen, touched, or felt), this definition has to include the documentation necessary to define the software and data in terms we can understand.

As you journey through these Guidelines, two major DoD software domains are discussed: weapon system software and management information system (MIS) software. Despite the different operational requirements of weapon system and MIS software, both domains perform the same functions in that they each collect, record, process, store, communicate, retrieve, and display information stored in or input to computers. The guidance you find here is applicable to the acquisition and management of ***all*** software-intensive systems — whether weapons systems or MIS. Differences in the development or management of software within the domains are the exception, not the rule, and will be brought to your attention as required. Software subcategories within the two domains include the following:

- **Weapon systems software**, comprised of,
  - Embedded software,
  - Command, control, and communication software,
  - Intelligence software, and
  - Any other software that is part of (or supports) a weapon system or its mission; and
- **Management information system (MIS)** software, including:
  - Information system resources (ISR) software,
  - Automated information system (AIS) software,
  - Information resource management (IRM) software, and
  - All other non-weapon system software.

Weapon System Software

Weapon systems include aircraft, ships, tanks, tactical and strategic missiles, smart munitions, space-launched and space-based systems, command and control (C2), and command, control, communications (C3), and intelligence (C3I) systems. **Weapon system software** is classified as embedded, C3, C3I, and all other software that supports

or is critical to the weapon system's mission. [ECSSP91] Examples of weapon system software are the Aegis radar and fire control system and the software on the B-2 bomber. B-2 bomber software, for instance, must oversee and coordinate avionics functions, surveillance, electronic countermeasures, smart munitions, and intelligence systems. A common weapon system software requirement is real-time processing. A **real-time** system performs several activities (or tasks), each of which must be completed by a specified deadline (e.g., radar signals, target positioning, weapon system status, etc.). Some of these deadlines may be hard (or critical) while some may be soft (such as those based on average performance). Missing a real-time hard deadline can result in catastrophic loss of system performance or even loss of life. [OBENZA94] Therefore, real-time software **reliability** requirements are often extremely demanding. "*The possible consequences of a worst-case failure in, say, a strategic weapon system dwarf those even for a nuclear power plant!*" [ZRAKET92]

Embedded Software

Embedded software is that which is specifically designed into, or dedicated to, a weapon system as an integrated part of the overall system. Embedded software functions as an integral part of the weapon system, and must be capable of satisfying the requirements for which it was designed or implemented; however, it does not readily support other applications without some form of modification. An example of embedded software is the software contained within the electronic circuitry of a smart weapon. The pilot can activate the *go-no-go* function allowing him to *fire-and-forget* his precision guided missiles. He cannot access, control, or modify the onboard software that governs the munition's radar, laser, and infrared guidance sensors or that activates the warhead. [HUEY91]

On the **F-16**, embedded software growth has been approximately one million lines-of-code per year since its avionics software evolved from the **F-111**. While the F-16's embedded software components themselves are very complex, they are only the tip of the total effort needed to develop and field complex, software-intensive systems, as illustrated in Figure 2-5. [ENGELLAND90]

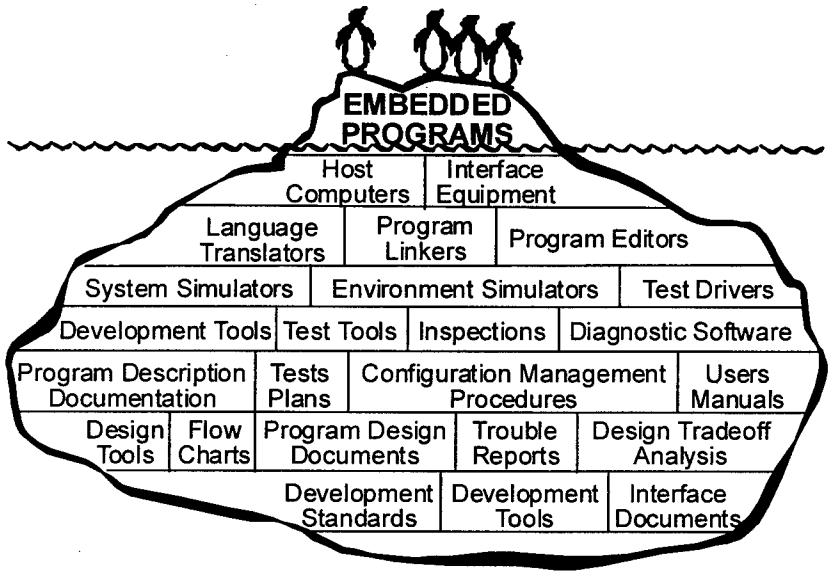


Figure 2-5 F-16 Embedded Software Iceberg

C3 Software

Command, control, and communications (C3) software is the component of weapon system software that communicates, assimilates, coordinates, analyzes, interprets information, and provides decision support to military commanders. Through advanced applications and computer technology, the C3 center aids commanders with their mission of exercising authority and giving direction to assigned forces. It provides instantaneous situational assessment, allowing for advantageous, timely positioning and decision-making. [JCS72]

Intelligence Software

Intelligence software, often combined with a C3 system (C3I), plays an important role in times of conflict and national security emergencies. It also maintains efficiency and responsiveness in day-to-day military operations. Intelligence software provides fast, reliable, secure information giving continuity to tactical or strategic operations under all conditions. It is designed to be dynamic and adapt to rapidly changing environments. This software has the

capacity for self-assessment through reliable warning functions that rapidly detect and react to threats or intruders. Intelligence software is found in command facilities and communications, surveillance, tracking and warning, navigation, and decision support systems. [WHITE80]

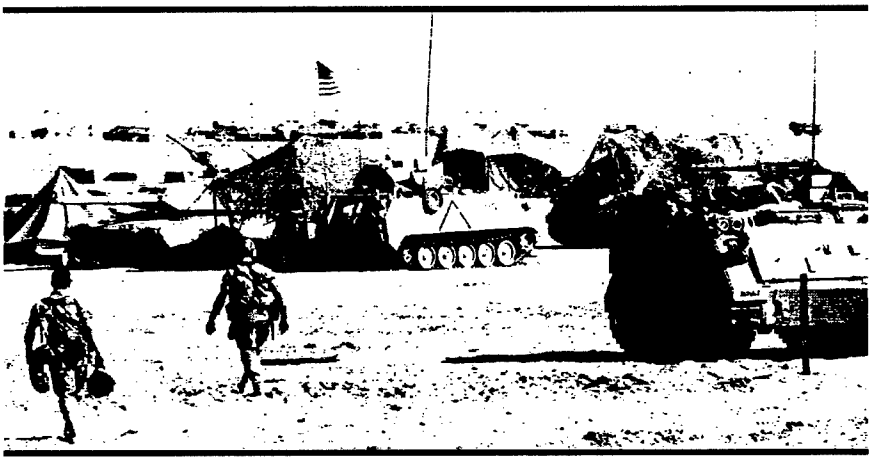


Figure 2-6 C3I Software Provides Secure Information to Tactical Operations

Other Weapon System Software

A variety of software associated with every weapon system exists that is not embedded, C3, or intelligence, but is nevertheless integral and absolutely essential. This software supports the weapon system and its mission. It includes software that performs mission planning, training, simulation, maintenance, battle management, system development, program management, scenario analysis, data reduction, configuration management, logistics, security, safety, quality assurance, and the testing of software and equipment. Examples of **other weapon system software** are the applications required to gather literally millions of data points generated during the ground and flight testing of any major developmental aircraft which is also required to aid in extensive data analysis and reduction. Figure 2-7 illustrates the concept of other weapon system software. [DSMC90]

The **Ballistic Missile Defense (BMD)** program illustrates the extreme range of functional performance requirements demanded of other weapon system software. BMD software controls surveillance,

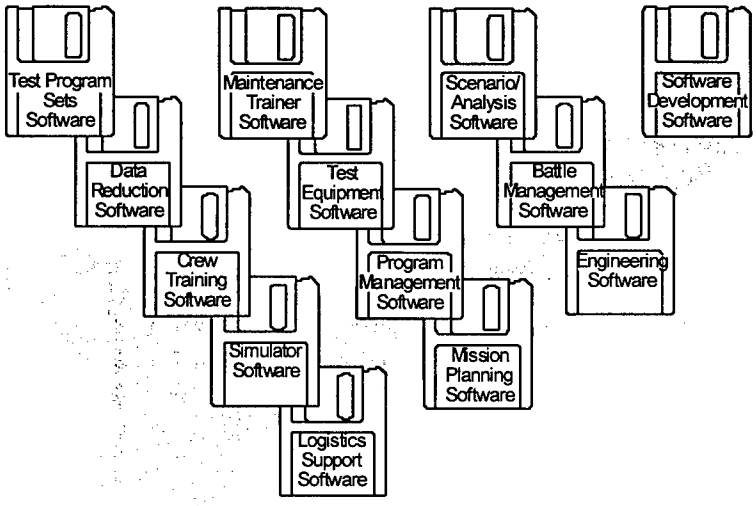


Figure 2-7 Other Weapon System Software (Not Embedded)

tracking, target detection and prioritization, weapons assignment, weapons control and guidance, system fault tolerance and fail-safe operations, network routing and management, security-access control, and damage assessment.

MIS Software (Non-Weapon System Software)

While embedded systems relate to and interface with physical world entities, MIS systems relate to the information world and can have upwards of 3,000 interfaces with other MIS systems. **MIS software** performs the functions of systems operations and support not associated with a weapon system. MIS manages administrative functions, such as accounting, payroll, finance, personnel, and similar information management activities. MIS software is also the main choice for such applications as inventory control, mapping, and equipment and maintenance scheduling. An MIS can access multiple, large databases of information where applications restructure existing data in a way that facilitates administrative operations or management decision-making. This category includes non-weapon system software, also called information system resource (ISR), automated information system (AIS), and information resource management (IRM) software. DoD relies heavily on commercially developed software for MIS applications. Security requirements, however, cut across both weapon system and MIS software domains.

A major initiative in the area of MIS is the **Global Combat Support System - Air Force/Base Level System Modernization Phase II [GCSS-AF (BLSM II)] program**. The goal of this program is to "*obtain total systems integration services and products to modernize standard Automated Information Systems (AISs) into integrated systems that are responsive to Air Force needs during times of war and peace.*" [GCSS95] With the RFP released in December, 1995, the program is in source selection and well on its way. *[For more information on the GCSS program, see the Standard Systems Group (SSG) Web page at the address listed in Volume 2, Appendix B.]*

ACQUISITION STREAMLINING: A National Imperative

Although software-intensive systems have forever changed the American military's view of the battlefield, they pose an unprecedented and monumental challenge to the acquisition corps tasked with supplying them to the warfighter. As Alvin and Heidi Toffler explain in their book, War and Anti-War, the future US military will be completely dependent on the third wave revolution of software-intensive information technology. However, the software needed to penetrate this third wave era fails to meet their book's basic mass production criteria for the second wave Industrial Age. They state that most software in use by our military is neither reproducible nor interchangeable. [TOFFLER93] As you learned in Chapter 1, *Software Acquisition Overview*, the ***weak software link*** in the automation chain must improve substantially if information is to flow seamlessly across the command, intelligence, logistic, and fire control networks of the Force XXI battlefield. As **Lieutenant General Otto Guenther**, Commander, US Army Communications, Electronics Command (CECOM), tells us, the future digitized Army *"will be driven by software...and we just aren't getting it right the first time."* [GUENTHER95]

In his *Report on the Bottom-Up Review*, **Secretary of Defense William J. Perry, Jr.,** defined the forces we need to fight and win two nearly simultaneous major regional conflicts, while at the same time conducting smaller-scale operations and maintaining a committed presence overseas. He also identified specific enhancements needed to bolster force capabilities, such as improvements in strategic mobility and the lethality of US firepower. [GAO95] In an interview with

Forbes magazine, **Admiral William A. Owens**, vice chairman of the Joint Chiefs of Staff, said the problem is getting more firepower out of less money by cutting DoD bureaucracy and pushing the four services to take full advantage of staggering advances in software-intensive technologies. He said ***we must find ways to be more efficient in using our procurement dollars***, because, *"if we keep an eye on where America's advantage lies, in high technology and quality people, leadership and training, then we will do well for our country and for our sons and daughters who have to fight our country's wars."* [OWENS95]

Decreasing Budgets — Increasing Software Demands

According to the **GAO**, DoD is the largest buying organization in the world, spending about $80 billion annually to research, develop, and procure weapon systems. DoD is accountable for over $1 trillion in assets with estimated outlays of $272 billion in Fiscal Year (FY) 1995 — approximately 50% of the federal government's discretionary spending. [GAO95] We have produced many highly effective weapon systems. There is, however, widespread agreement — among DoD, the defense industry, and the Congress — that *[as you also learned in Chapter 1, Software Acquisition Overview]* our process for determining weapon system requirements and procuring software-intensive-systems often is costly and inefficient.

Shrinking procurement funds, dwindling forces, and expanding missions are compounding these challenges for acquisition managers with post-Cold War defense structure and budget downsizing. Over the past 10 years, the number of personnel in uniform has declined by 28% and a significant number of Army divisions, Air Force wings, and Navy ships have been removed from active service. During the same period, the defense budget has fallen from $391 billion to $252 billion (in constant 1995 dollars) — a reduction of 35%. [GAO95] To emphasize the importance of software to DoD in light of budget declines, in FY92, DoD spent over $35.2 billion on software-intensive systems, $29.1 billion, or 83%, of which was for software alone. **Lieutenant General Carl G. O'Berry**, former Air Force Deputy Chief of Staff (command, control, communications, and computers), tells us the FY95 software budget reached $42 billion — a 31% increase in just three years.

DoD's New Order of Acquisition

> *Laws and institutions must go hand in hand with the progress of the human mind. As that becomes more developed, more enlightened, as new discoveries are made, new truths disclosed, and manners and opinions change with the change of circumstances, institutions must advance also, and keep pace with the times.*
> — Thomas Jefferson, July 12, 1816

In a speech before the Armed Forces Communications and Electronics Association (AFCEA) conference, **Senator William S. Cohen** remarked that, *"If we are to strive for a more efficient government, assess new threats on the horizon, and achieve what is truly possible in the areas of pilotless aircraft, electronic warfare, and battlefield technology management, we are going to have to reform how we buy and use information technology."* [COHEN95] To survive in the global marketplace, private sector industries know they must constantly modernize and upgrade their management practices to make themselves more productive and reduce costs. Today, many federal agencies face the same reality. DoD is a prime example of an agency facing the challenge of streamlining for efficiency and lowering costs while maintaining quality. Over the past several years, DoD has been consumed with the need to reform how we spend our declining budget dollars. The bottom line is we have to improve the way we provide the warfighter — in the trenches, in the cockpit, and on the bridge — the software they need, that works, is affordable, and delivered on time. [BROWN95]

Colleen Preston, Deputy Under Secretary of Defense (acquisition reform), explained that acquisition reform is imperative because the system is antiquated and cannot keep up with modern day advances in technology. She said, *"This is the 10th year of a declining defense budget. Technology is changing so rapidly that the [acquisition] system can't keep up. Look at information systems technology, which turns over on an average of every 18 months. Yet, to process a simple Request for Proposal, not using small purchase procedures, takes an average of 90 days; a negotiated procurement, an average of 210 days; and a complex services contract to support one of our program management offices, an average of 300 days. We can't even get on contract before technology is obsolete."* [PRESTON95]

Acquisition Reform Working Group

In 1993, a coalition of eight defense and aerospace-related organizations formed an **Acquisition Reform Working Group** to study the defense acquisition reform issue from a suppliers' perspective. (The working group included members from the Aerospace Industries Association, American Defense Preparedness Association, American Electronics Association, Contract Services Association, Electronic Industries Association, National Security Industrial Association, Professional Services Council, Shipbuilders Council of America, and the US Chamber of Commerce.) The group determined that the ***countless non-value-added requirements DoD imposes on its contractors adds 20% to 50% to DoD's cost of doing business*** — limiting its buying power and precluding thousands of firms from doing business with the federal government. The group concluded that, through excessive regulation and micro-management, DoD has jeopardized the financial health of the defense industry as a whole. [SMITH94] **C. Michael Armstrong,** Chairman and CEO, Hughes Aircraft Company, said that his company experienced cuts of up to 30% with defense spending drawbacks. He exclaimed that,

> *You're all familiar with the recently completed Bottom-Up Review. In the defense industry we have been involved in what I might call the Belly-Up Review — because that's the consequence of failure.* [ARMSTRONG93]

Acquisition Reform: A Mandate for Change

In **Defense Secretary Perry's** February 1994 report, ***Acquisition Reform: A Mandate for Change***, he also recognized that American companies most dependent on defense business are laying off hundreds of thousands of workers. These jobs will be gone for good unless former defense-only companies can convert to manufacturing commercial products. If DoD does not aid in this conversion, by adopting procurement practices that encourage commercialization, it will lose access to the industrial base upon which it relies for technological superiority.

Perry explained that for years DoD pioneered technological advances in many areas — but today, the tables have turned. Commercial technology advancements are outpacing DoD-sponsored efforts in many sectors key to military superiority (e.g., computers, software,

integrated circuits, communications, and advanced materials). From R&D to practical application and production, DoD simply takes too long. The design cycle for commercial technology is approximately 3-4 years; in DoD it is 8-10 years. Many of the advanced technologies DoD implements are grossly obsolete before even fielded. Perry reasoned that to maintain our military superiority, we must gain access to commercial technologies more quickly and more economically than other countries.

Perry concluded that DoD acquisition reform coincides with our most important national goals: saving the taxpayer money; reinventing Government; strengthening our military; and improving our economy. To meet these goals, DoD must:

- Rapidly acquire commercial and other state-of-the-art products and technology from reliable suppliers who employ the latest manufacturing and management techniques;
- Assist in the conversion of US defense-unique companies to dual-use production;
- Transfer military technology to the commercial sector;
- Preserve defense-unique core capabilities (e.g., submarines, armored vehicles, and fighter aircraft);
- Integrate, broaden, and maintain a National Industrial Base sustained by commercial demand but capable of meeting DoD needs;
- Adopt the business processes of world-class customers and suppliers (including processes that encourage DoD suppliers to do the same); and
- To the maximum extent practicable, stop placing government-unique terms and conditions on its contractors.

Federal Acquisition Streamlining Act of 1994

On October 13, 1994, **President Bill Clinton** signed the **Federal Acquisition Streamlining Act of 1994**, Public Law 103-355, which was designed to create a more equitable balance between government-unique requirements and the need to lower the Government's cost of doing business. According to Norm Augustine, former Under Secretary of the Army and Lockheed Martin Corporation CEO, this legislation is *"the first successful initiative in memory to reform the much-maligned defense acquisition process."* [AUGUSTINE95] **Dr. Paul Kaminski**, Under Secretary of Defense (acquisition and technology), claims that, *"The Acquisition Streamlining Act of 1994 is the most significant change in law*

affecting procurement in five decades. It will transform the way we buy goods and services." [KAMINSKI94]

The Act emphasizes increasing Government-wide reliance on the use of commercial practices, goods and services; streamlining the rules and regulations that govern high-volume, low-dollar contracting activities; and increasing contracting opportunities for small business. [DRELICHARZ94] The Act prescribes that the Government must first consider the purchase and use of commercial items, and then nondevelopmental items (NDI). Only when it is determined that neither of these items are available can the Government consider procuring specially-designed, government-unique items. The Act lifts many formerly rigid provisions and allows DoD to follow business practices so that its suppliers do not need separate production lines — one for defense and one for commercial products. As Kaminski noted, *"We have turned the system upside down...now we must tell the contractor what we need the system to do, not how to do it."* [KAMINSKI94] The Act also establishes a **Federal Acquisition Computer Network** that contains an automated list of what the Government wants to buy. Suppliers will be able to submit proposals electronically, eliminating paper solicitation and paper contracts.

MilSpec and MilStd Reform

To understand the acquisition reform taking place in DoD, one must understand that its fundamental purpose is to enhance and unify the commercial and defense industrial base by applying the most modern industrial products, processes, and practices to our acquisitions. ***Remember, cost is the key driver in acquisition reform initiatives.*** For software, this includes adapting the most modern methods and principles of software engineering *[discussed in Chapter 4, Engineering Software-Intensive Systems].* In Chapter 13, *Contracting for Success*, many strategies for cutting software acquisition costs are introduced. Increased user involvement throughout the development process, reuse, simplicity of design, open systems architectures, peer inspections, metrics, prototyping/ demonstrations, and iterative/evolutionary system developments are all ways to reduce acquisition costs.

As discussed above, where DoD was previously the driving force, the commercial sector is now the catalyst behind the development of many high-tech industries. [GARCIA94] Thus, DoD must ***start***

doing business more like business by taking advantage of savings found in **commercial-off-the-shelf (COTS)** for all those software functions which can be fulfilled by those products. The shift to COTS products will be easier for MIS than for weapon systems applications, and is going to require a change in the level of detail found in requirements specifications. The acquiring organization must be willing to back off on over-specifying requirements, and actually reclassify many requirements as preferences driven by cost tradeoffs, the same way business does.

It also means moving away from requiring strict adherence to military specifications (MilSpecs) and standards (MilStds) in our acquisitions. Granted, many of the MilSpecs and MilStds provide common, detailed, and precise descriptions essential to the Government and its suppliers in the execution of a contract. But, as the February 1994 ***Report of the Industry Panel on Specifications and Standards*** states, just as many of them are misapplied, obsolete, redundant, or unnecessarily restrictive. All these non-value-added requirements have increased the price the Government pays for goods and services.

In April 1994, the Office of the Under Secretary of Defense (acquisition and technology) issued the ***Process Action Team Report on Military Specifications and Standards***. This report concluded that by requiring contractors to comply with rigid military specifications and standards, DoD is paying a *defense-unique* premium for the goods and services it buys. DoD has also been fighting an uphill battle in keeping some 31,000 MilSpecs and MilStds current with fast-paced changes in technology, the military mission, and threat environment. The report states that as DoD's budgetary and manpower resources are reduced, there is little hope MilSpecs and MilStds can be kept technically up-to-date with commercial practices, products, or standards.

Specifications & Standards — A New Way of Doing Business

In response to these and related concerns, **Secretary Perry** issued the June 1994 policy memorandum, **Specifications & Standards — A New Way of Doing Business.** *[See Volume 2, Appendix C.]* Perry explained that the backbone of our military superiority rests in our ability to field the most superior, advanced technology. To meet future DoD needs, we must increase our access to the high-tech commercial sector by reducing the requirements for MilSpecs and

MilStds in our acquisitions. DoD must make greater use of performance, commercial specifications, and standards in the procurement of all new systems, major modifications, and upgrades to current systems and nondevelopmental and commercial items. *[See Volume 2, Appendix D for a list of commercial software standards.]* If the use of a non-government standard is not acceptable or cost effective, a MilSpec or MilStd can be used, with an appropriate waiver from the **Milestone Decision Authority (MDA)**.

Performance specifications. According to the **Defense Standards Improvement Council (DSIC)**, a performance specification is one that states requirements in terms of required results with ***criteria for verifying compliance*** — it does not describe how to go about achieving required results. It defines the item's functional requirements, the environment in which it must operate, and ***interface and interchangeability characteristics***. There are four types of performance-based specifications:

- **Commercial item descriptions (CIDs).** Guidance for CIDs is found in the **GAO Federal Standards Manual**, **DoD 4120.3-M**, *Defense Standardization Manual*, and in the **DoD SD-2**, *Buying NDI.* CIDs describe requirements in terms of function, performance, and essential form and fit requirements. Many MilSpecs are cited as future candidates for CIDs.
- **Guide specifications. DoD 4120.3-M** gives direction for guide specifications which standardize common functional and performance requirements for similar systems, subsystems, equipment, and assemblies. The format of the guide specification forces the user to tailor the document to their specific application. Many MilSpecs are cited as candidates for guide specifications.
- **Standard performance specifications. MIL-STD-961C,** *Preparation of Military Specifications and Other Documents*, is being expanded to include direction on the content and format of performance specifications for use in multiple applications. Many MilSpecs are cited as future candidates for performance specifications.
- **Program-unique specifications**. As of this publication, there is no formal guidance for program-unique specifications. A rule of thumb is that they should be described in terms of *"performance."*

NOTE: The DoD Index of Specifications and Standards (DoDISS) currently lists approved CIDs and guide specifications and has a section on standard performance specifications. *The Acquisition Streamlining and Standardization Information SysTem (ASSIST)* is an on-line database (available to all DoD personnel) for tracking decisions about the MilSpec initiatives and standardization documents listed in the DoDISS. *[To subscribe, contact the Defense Printing Office at DSN 442-6257. Also, you can view the DoDISS on the Web (see Volume 2, Appendix B for the Web address). For current MilSpec and MilStd reform efforts, see the Web address in Appendix B for OSD (acquisition and technology) MilSpec and standards reform information.]*

The **Defense Standards Improvement Council (DSIC)** has been tasked with identifying those software MilSpecs and MilStds, for which there are no acceptable commercial alternatives, which should have a *blanket waiver* to the **Perry Memo**. Those currently tagged for blanket waiver review are performance specifications, interface standards, standard practices, reference standards, and data acquisition standards/specifications. Until these blanket waivers are approved, you are advised to be intimately familiar with the current MilSpecs and MilStds relating to software development (especially those pertaining to *process*). If you determine there are specific MilSpecs or MilStds that will significantly reduce your acquisition risk, you can *"cite"* them in your solicitation. This citation is *"for guidance only"* — without the need for a waiver. It acts as a means to communicate to bidders the types of requirements to which they should propose.

Remember, bidders are not restricted by Perry's Memo from proposing the use of MilSpecs and MilStds, which in some cases, are the commercial standard. However, you need to be cautioned if bidders do not propose adequate commercial standards or the software MilSpecs and MilStds you cite as examples. If this occurs, it is strongly recommend that ***you obtain the waivers*** for those software MilSpecs and MilStds that effectively mitigate your acquisition risk. Also, if your solicitation is for the maintenance or enhancement of legacy software, you can require performance-based MilSpecs and MilStds (without a waiver) if no design change is required. *[The software MilSpecs and MilStds with which you should be familiar are listed in Volume 2, Appendix D. Additional information can be obtained through the Web addresses listed in Volume 2, Appendix B.]*

MIL-STD-498, *Software Development and Documentation*

MIL-STD-498, *Software Development and Documentation,* is the preferred standard for all DoD software development. It has a blanket waiver exemption to the Perry Memo from the **Air Force Standards Improvement Executive,** the **Navy Standards Improvement Executive**, and on a *case-by-case* basis by the Army. It has been incorporated into an **International Standards Organization (ISO)** standard (ISO 12207) which has a US implementing standard, IEEE 1498. MIL-STD-498 supersedes DoD-STD-2167A, DoD-STD-7935A, and DoD-STD-1703 (NS). ***It defines a set of activities and documentation suitable for all software-intensive systems: weapon systems, C3 systems, and MIS.*** MIL-STD-498 defines a process to follow and the skill level required for its implementation. It is designed to be tailored to the type of software to which it is applied. *[See MIL-STD-498, Application and Reference Guidebook, for information on applying the standard. See MIL-STD-498, Overview and Tailoring Guidebook, for guidance on tailoring.]*

MIL-STD-498 is compatible with incremental and evolutionary development models, non-hierarchical design methods, and computer-aided software engineering (CASE) tools. It provides: alternatives to (and more flexibility in) documentation preparation; requirements for reusable software; guidance for software management indicators, metrics, and software supportability; and clear links to systems engineering. It explains how systems designs are partitioned into units using appropriate development methods compatible with Ada. Behavioral, architectural, and detailed designs for configuration items, as well as data bases, are also defined. Contractors electing to use this standard are required to comply with security and privacy requirements. The standard describes those software development activities which should be addressed in the contractor's **Software Development Plan (SDP).** How the contractor implements these activities, however, is at their discretion. The items to include in the SDP are:

- **Software development methods** (e.g., the use of systematic, documented methods),
- **Software products standards** (e.g., a standard must be developed/adopted and applied to requirements, designs, code, test cases, test procedures, and test results),

- **Reusable software products** (e.g., incorporation and development of reusable products),
- **Handling of critical requirements** (e.g., assurance of safety, security, privacy, and other critical requirements),
- **Computer hardware resource utilization**,
- **Recording rationale** (e.g., key decisions made in specifying, designing, implementing, and testing software), and
- **Government review access** (e.g., access to developer and subcontractor facilities, including software engineering and test environments).

DoD Policy on the Use of Ada

As early as 1974, DoD attempted to change *business-as-usual* by curbing runaway software spending. It did this by sponsoring the development of a standardized programming language, **Ada**. Unfortunately, until Congress mandated its use for all DoD software developments funded in FY91 through the **1991 Defense Appropriations Act**, Ada's benefits were not fully appreciated and waivers to its use were commonplace. A costly example was the **C-17** program, the most software-intensive air transport ever built. Started before the DoD mandate, programming was allowed in six different software languages. *[See Chapter 1, Software Acquisition Overview, for a discussion on C-17 software problems.]*

On August 26, 1994, **Noel Longuemare**, acting Under Secretary of Defense (acquisition and technology), and **Emmett Paige, Jr.**, Assistant Secretary of Defense (command, control, communications, and intelligence) issued the ***DoD Policy on the Use of Ada.*** This was in response to the June 1994 Perry Memo *[discussed above]*. The Ada memorandum was issued to clarify that the Ada requirement does not conflict with the need to obtain a waiver for the use of MilStds. As an ANSI, ISO, and FIPS standard, Ada 95 is a commercial and international standard that must be cited in your RFP as a required best practice for bidders. If other than Ada is proposed, such proposals are required to provide strong justification that *overall life cycle costs* (not just development costs) will be less than with the use of Ada. *[For more information, see the new DoD 5000.2-R and revised DoDD 3405.1.]*

CHALLENGE! **The documentation and sometimes special (i.e., unique to DoD) processes dictated by the MilSpecs and MilStds are estimated to add 30% to the cost of major software-intensive systems acquisitions. The Perry Memo directs you to forego the use of MilSpecs and MilStds in deference to *"normal commercial practices."* Your challenge is to assure that the sound software engineering principles and practices described in the MilSpecs and MilStds, and especially in the remaining pages of these Guidelines, are embodied in the *"normal commercial practices"* of the contractors/ developers you employ. In this regard, see in particular the discussion in Chapter 7, *Software Development Maturity.***

C4I FOR WARRIOR

At the 1995 Software Technology Conference, *Architecting the Information Highway for the Warrior*, **Rear Admiral John G. Hekman,** former commander of the Naval Information Systems Management Center (NISMC), explained that because *"all data is tactical, this requires a paradigm shift in how we architect and manage* [our software-intensive resources.] *We must migrate towards the ultimate goal of complete data compatibility... and single interoperable solutions across all domains. Our vision... for a joint common tactical picture...is for an open system community where every solution is interoperable, seamless, and transparent to the user."* [HEKMAN95]

The need for interoperability, portability, reusability, and flexibility to accommodate changing threat environments was painfully apparent during the Gulf War. Rapid force buildup during **Operation Desert Shield** and continuous movement of forces during **Operation Desert Storm** greatly taxed our communications systems. Before coalition forces could fight together, they first had to talk together. Lacking a comprehensive joint architecture, forging together an integrated international communications network was close to impossible. Significant interoperability issues surfaced among the friendly nations in the attempt to hurriedly piece together dissimilar equipment. Interoperability was also an issue within the US military itself, which

relied on several generations of incompatible analog and digital communication systems. [WENTZ92]

To be useful, a software architecture must first include provisions for change, and second, be controllable and maintainable throughout the system's life. As **Lt. Gen. Gordon E. Fornell** (retired), former Commander, USAF Electronic Systems Center, stated in a cover letter to the report, *Process for Acquiring Software Architectures*,

> *The more substantial portion of maintenance cost is devoted to accommodating functional changes to the software necessary to keep pace with changing user needs. Based on data we had reviewed, we also noted that systems with well-structured software were much better able to accommodate such changes.* [FORNELL92]

DoD now requires the acquisition of **open system environments (OSEs)** for *all* information systems, C3I systems, and strongly endorses this approach for avionics and embedded software applications to the maximum practical extent. To enact DoD mandates about open systems, a new Joint Chiefs of Staff plan calls for the reorganization of DoD's C4I infrastructure. The plan, called ***"C4I for Warrior,"*** provides policy guidance on DoD information management into the 21st century and eliminates service-specific systems. *"The goal is 100% interoperability,"* explained **Major General Albert J. Edmonds** while JCS Deputy Director for C4I Support. [EDMONDS93]

Open Systems

The definition of an **open system** (e.g., IEEE POSIX) is one that implements open specifications for interfaces, services, and supporting formats. Open systems assure, provide, and are the basis for interoperability. An open system enables properly engineered components to be utilized across a range of systems. Minimal changes are required for the components to interoperate on local and remote systems and to interact with users in a style that facilitates portability. An open system is characterized by:

- Well-defined, widely used, non-proprietary interfaces/protocols;
- Use of standards developed/adopted by industry-recognized standards bodies;

- Definition of all aspects of system interfaces to facilitate new or additional capabilities for a wide range of applications; and
- Explicit provision for expansion or upgrading through the incorporation of additional or higher performance elements with minimal negative impact on the existing system. [OSA92]

An additional characteristic to consider is the open system's "*popularity*." This is an important characteristic, and is even addressed in Section 11 of the FAR, which allows popularity as one factor in the evaluation of an open system. [HOROWITZ95]

An **open system architecture** means the software is portable in the sense that its use is not dependent on specific hardware platforms or operating system software. The major benefits of an open system architecture are: (1) costs are reduced through information sharing, interoperability, and portability; (2) the possibility of using commercially-available software, or reusing software developed for other systems, is increased; and (3) change is easier to track throughout the software life cycle. **FAR** and **DoD regulations** governing hardware and software purchases emphasize the need for competitive procurements, discourage sole-sourcing, and encourage acquiring portable software that operates on hardware from different vendors. In an interview with *Government Computing News*, **Emmett Paige Jr.**, Assistant Secretary of Defense for C3I, commented on the interoperability issue.

> *The big issue with me is open systems. Standards bodies are not the issue, although we need standards. If no one is pushing for them, we won't get them. If you don't have them, you won't have anything but proprietary systems.* [PAIGE93]

Technical Architecture Framework for Information Management (TAFIM)

In 1993 DoD issued the **TAFIM** in final draft as the standard to which all DoD information systems architectures are to be built. It provides a single framework for the integration of all DoD information systems, expanded opportunities for interoperability, portability, and scalability, and enhances the manageability of DoD information resources. It applies to all DoD information system ***technical architectures*** at every organizational level and environment (i.e., tactical, strategic, sustaining base, interfaces to weapon systems).

The TAFIM's goal is to achieve the evolution of a DoD-wide technical infrastructure by providing services, standards, design concepts, components, and configurations for specific system architectures. It gives acquisition guidance, addresses transitioning to an open system environment, and is a living document that will evolve as technology evolves. It is maintained by **DISA** and is available through DISA's Online Standards Library and through the **Defense Technical Information Center (DTIC).**

The latest version of the TAFIM was put into effect by the March 30, 1995 policy memorandum, *Technical Architecture Framework for Information Management (TAFIM)*, Version 2.0 *[see Volume 2, Appendix C],* signed by **Emmett Paige, Jr**. This memorandum made TAFIM compliance mandatory for all new DoD information systems development and modernization programs and for all evolutionary changes to migration systems. The TAFIM adopts the foundation of the IEEE POSIX P1003.3.0 Working Group as reflected in their *Draft Guide to the POSIX Open Systems Environment (POSIX.0)*. TAFIM, Version 2.0 consists of seven volumes, as illustrated on Figure 2-8.

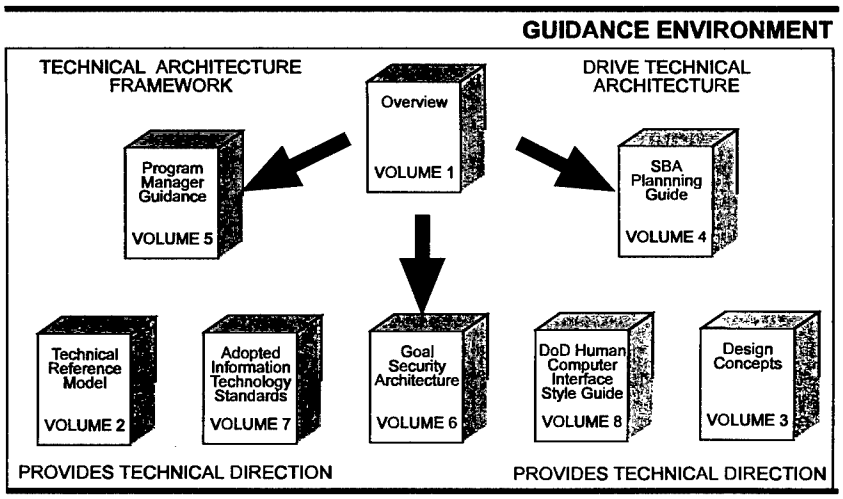


Figure 2-8 TAFIM Contents [DISA95]

- **Volume 1, Overview**. The Overview provides a detailed description of the technical architecture framework concept and guidance on using the technical architecture in information systems development. It defines the relationships between information technology acquisition and DoD's reuse programs. The TAFIM

contains five general layers/components: Application Software Entity, Application Programmer Interface (API), Application Platform Entity, External Environment Interface (EEI), and External Environment.

- **Volume 2, Technical Reference Model (TRM)**. As illustrated on Figure 2-9 (below), the TRM for information management describes services and interfaces and provides a profile of standards.
- **Volume 3, Architecture Concepts and Design Guidance**. This volume provides descriptions of layers, an interface service model, design concepts, and architectural guidance.
- **Volume 4, Standards-Based Architecture (SBA) Planning Guide**. The SBA Planning Guide gives direction for application of the **Technical Architecture Framework** and provides a standard methodology for the development of technical architectures, as illustrated in Figure 2-10 (below). The starting point for SBA planning is the **Architectural Modeling Framework**, as illustrated on Figure 2-11 (below). It breaks the SBA into various components as they relate to strategic drivers. Architectural principles reflect boundaries and requirements as dictated by DoD policy. Functional requirements are addressed in work, application, and information architectures. Existing technical requirements and the evolution/migration to future technologies are addressed in the technical architecture.
- **Volume 5, Program Manager Guidance**. Program manager guidance consists of lists of support services available from DISA for users of the TAFIM. It provides words to include in an RFP to require the use of the TAFIM by contractors. It also lists references for guidance on business process re-engineering, data modeling, and standardization.
- **Volume 6, DoD Goal Security Architecture**. DoD security goals specify security principles and target security capabilities for guiding the system security architecture, as illustrated in Figure 2-12 (below).
- **Volume 7, Adopted Information Technology Standards (AITS)**. Standards promote interoperability and portability and are the foundation for open systems. The AITS identifies those standards selected for use DoD-wide. The **Information Technology Standards Guidance (ITSG)** document provides detailed AITS guidance, rationale for AITS selection, and comprehensive descriptions of functions, standards, deficiencies, and alternatives. It explains how an open systems environment must be defined in terms of technical requirements prior to the selection of standards to fit those requirements. The ITSG also aids

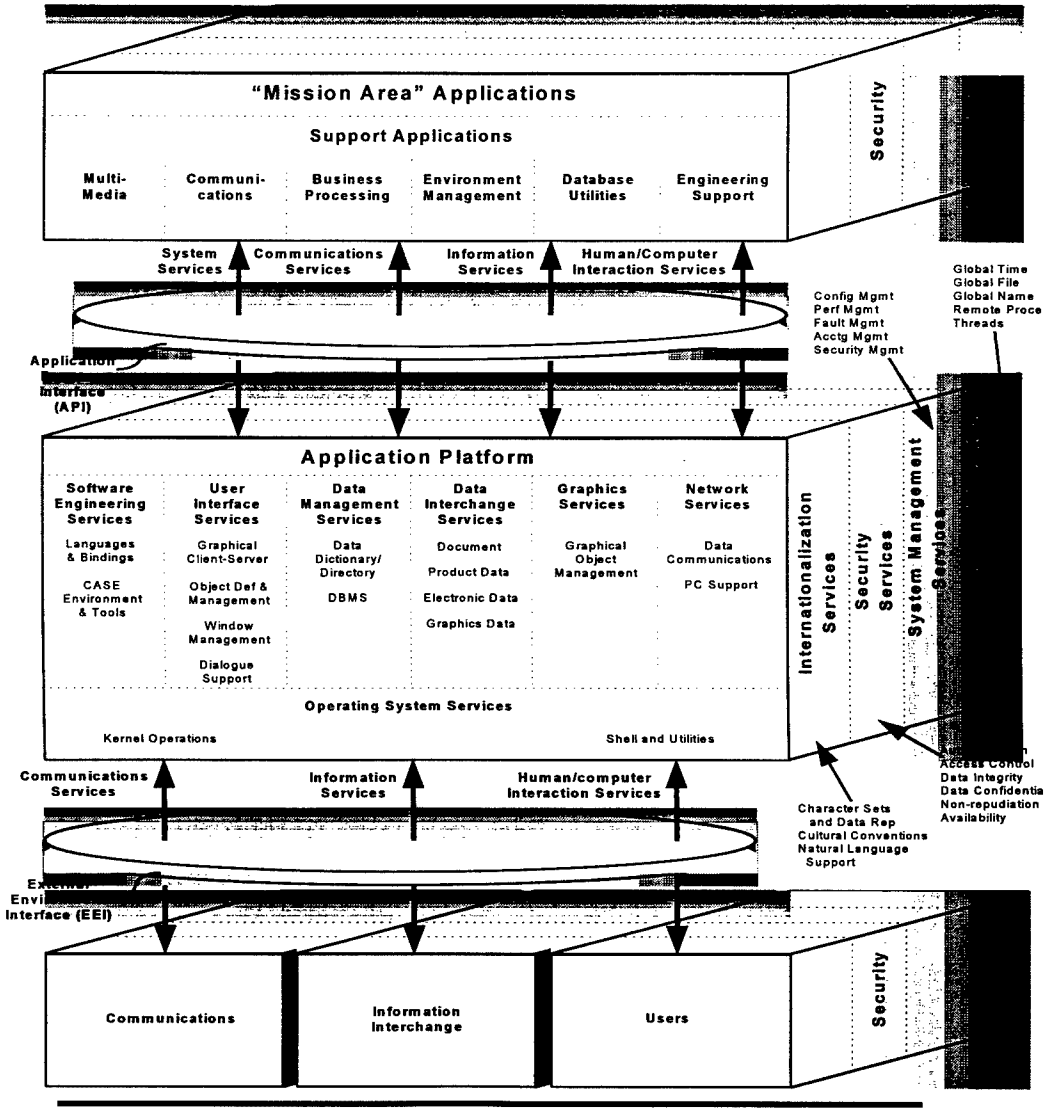


Figure 2-9 TAFIM Technical Reference Model [SENTERS95]

in the selection of suitable standards and lists only those needed for procurement.

- **Volume 8, Human Computer Interface (HCI) Style Guide.** The HCI Style Guide gives guidance on how a user interface should look and feel, as illustrated on Figure 2-13 (below).

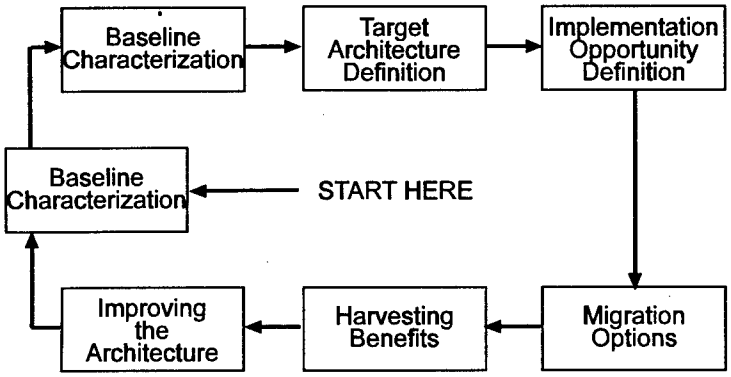


Figure 2-10 Technical Architecture Framework [DISA95]

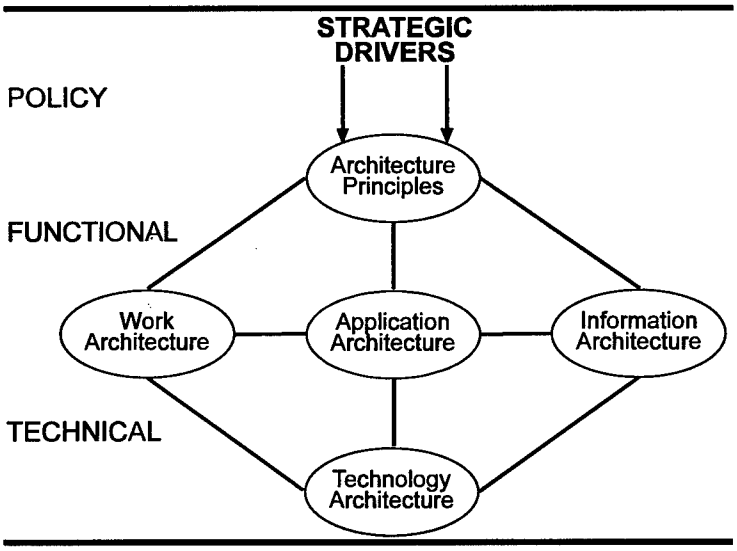


Figure 2-11 Architectural Modeling Framework [DISA95]

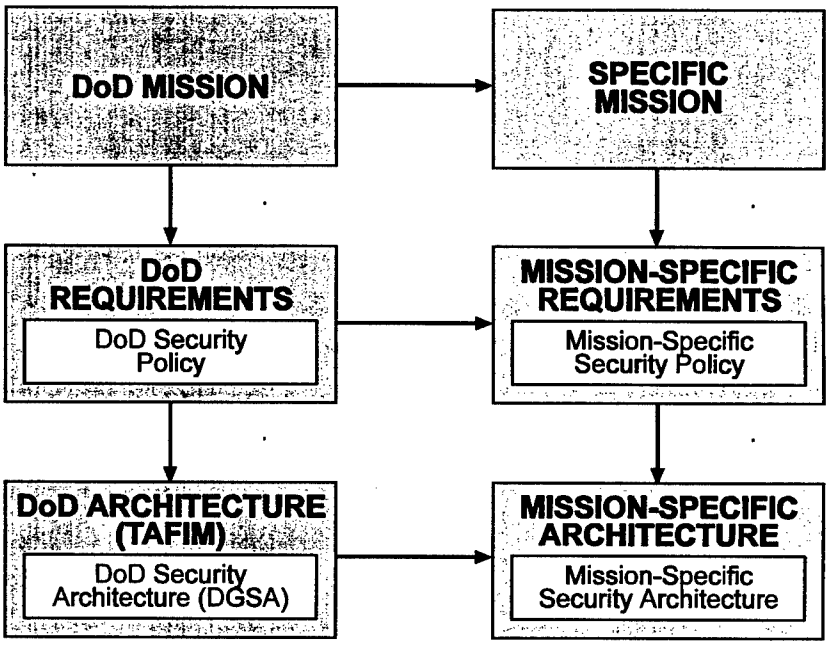


Figure 2-12 DoD Goal Security Architecture [DISA95]

SOFTWARE BEST PRACTICES INITIATIVE

In response to the **June 1994 Perry Memo,** the **Software Best Practices Initiative** was instituted by **Noel Longuemare** and **Emmett Paige Jr.,** *[see Volume 2, Appendix C].* This initiative was based on the fact that many effective practices exist for managing software — both in industry and Government. However, their use and understanding are not widespread within DoD software acquisition programs. Seven panels studied successful software programs in the public and private sectors to determine those practices characteristic to all programs and significant leverage items for success. The goals of the initiative are to:

- Focus the acquisition community on employing high-leverage software acquisition management practices,
- Enable managers to focus more on their software development programs than on the regulations for purchasing those systems, and
- Change practices that deviate from the ***corporate and program cultures*** outside of DoD. [JONES94]

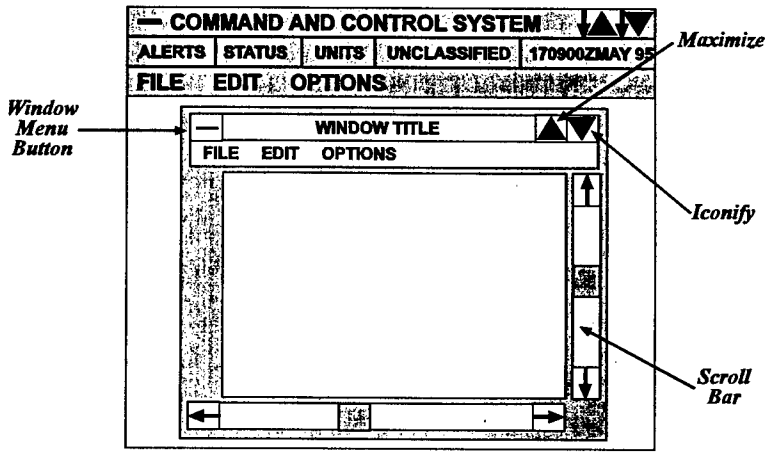


Figure 2-13 Human Computer Interface (HCI) Style Guide [DISA95]

The **Software Best Practices Initiative** represents the collective efforts of nearly 200 software development and maintenance expert practitioners from the commercial and government world, industry leaders, software visionaries, and methodologists. Table 2-1 lists the ***Worst Practices*** identified and repackaged in the form of **Project Caveats**:

PROJECT CAVEATS
Don't use schedule compression to justify usage of new technology on any time critical project.
Don't have the government mandate technological solutions.
Don't specify implementation technology in the RFP.
Don't advocate use of unproven *"Silver Bullet"* approaches.
Don't expect to recover from any substantial schedule slip (10% or more) without making more than corresponding reductions in functionality to be delivered.
Don't put items out of project control on the critical path.
Don't expect to achieve large positive improvements (10% or more) over past observed performance.
Don't bury all project complexity in the software (as opposed to hardware).
Don't conduct the critical system engineering tasks (particularly the hardware/software partitioning) without sufficient software expertise.
Don't believe that formal reviews provide an accurate picture of the project. Expect usefulness of formal reviews to be inversely proportional to the number of people attending beyond five.

Table 2-1 Software Acquisition Worst Practices

NOTE: Listed in Chapter 16, *The Challenge*, "*What to Do With a Troubled Program*?" are a set of *Breathalyzer* questions, integral to the Best Practices, that can be used to identify those programs most likely to fail unless fundamental corrective management action is taken.

The **Best Practices** observed to be most frequently and successfully used in industry, deemed essential for DoD software development programs, include the following *[as published in NetFocus the newsletter of the **Software Program Managers Network**, April 1995].*

- **Formal risk management**. *[See Chapter 6, Risk Management, for an in-depth discussion.]* The discipline of risk management is vital to the success of any software effort. This discipline requires as a minimum:
  - Appointing a Risk Officer whose primary responsibility is risk management,
  - Compiling a database for all non-negligible risks,
  - Preparing a profile for each risk including: probability, cost impact, schedule impact, earliest expected visible symptom, action plan(s) to be invoked upon detection, and weighted contribution to risk reserve,
  - Continuously updating and monitoring risk plans to account for new potential and manifest risks,
  - Managing and controlling the integrity of information,
  - Managing technical, programmatic, supportability, cost, and schedule risks for the life of the system, and
  - Chartering all program personnel to be risk identifiers.
- **Agreement on interfaces**. *[See Chapter 14, Managing Software Development, for an in-depth discussion.]* To address the chronic problem of vague, inaccurate, and untestable specifications, a baseline user interface should be agreed upon across affected areas before beginning implementation activities. This user interface must be developed and maintained as an integral part of the system specification, and updated only as necessary during development. Included in the user interface description are:
  - Full census of inputs and outputs across the system boundary, and
  - Definition of each input and output down to the data element level.

- For those programs developing both hardware and software, a separate software specification should be written with an explicit, complete interface description consisting of:
  -- Full census of inputs and outputs between software and all externals, and
  -- Definition of each input and output down to the data element level.

- **Metric-based scheduling and tracking**. *[See Chapter 8, Measurement and Metrics, for an in-depth discussion.]* Statistical quality control of costs and schedules should be maintained. This requires early calculation of size metrics, projection of costs and schedules from observed empirical patterns of past results, and tracking of program status through the use of captured result metrics. Use of a parametric analyzer or other automated projection tool is also recommended to track against initial and current cost/ schedule baselines, and the number of changes to the initial cost/ schedule baseline.
- **Defect tracking against quality targets**. *[See Chapter 15, Managing Process Improvement, for an in-depth discussion.]* Defects should be tracked formally during each program phase. Configuration management enables each defect to be recorded and traced to its removal. In this approach, there is no such thing as a *private defect*, i.e., one detected and removed without being recorded. Initial quality targets (expressed, for example, in defects per function point) as well as calculations of remaining or latent defects are compared to defects removed counts to track progress during testing activities.
- **Program-wide visibility of program plan and progress versus plan**. *[See Chapter 15, Managing Process Improvement, for an in-depth discussion.]* The core indicators of program health or dysfunction should be made available to all program participants. Anonymous feedback should be encouraged to enable bad news to move up and down the program hierarchy.
- **Configuration management (CM).** *[See Chapter 15, Managing Process Improvement, for an in-depth discussion.]* The discipline of CM is vital to the success of any software effort. This discipline requires as a minimum:
  - Control of shared information,
  - Control of changes,
  - Version control,
  - Identification of the status of controlled items (e.g., memos, schedules), and
  - Reviews and audits of controlled items.

- **Inspections, reviews, and walkthroughs**. *[See Chapter 15, Managing Process Improvement, for an in-depth discussion.]* Peer reviews should be conducted at all design levels (particularly detailed designs), on code prior to unit test, and on test plans.
- **Binary quality gates at the inch pebble level**. *[See Chapter 15, Managing Process Improvement, for an in-depth discussion.]* Completion of each task in the lowest level activity network needs to be defined by an objective binary indication. These completion events should be in the form of *"gates"* which assess the quality of the products produced, or the adequacy and completeness of the finished process. Gates may take the form of technical reviews, completion specific test sets which integrate or qualify software components, demonstrations, or program audits. The binary indication is meeting a predefined performance standard (e.g., defect density). Activities are closed only upon satisfying the standard, with no partial credit given. *[NOTE: Frequent software development organization product and process reviews are encouraged, and should not necessarily be directed or even attended by members of the government acquisition staff.]*
- **People-aware management accountability**. *[See Chapter 13, Contracting for Success, for an in-depth discussion.]* Both acquisition management and developer management must be accountable for staffing qualified people (i.e., those with domain knowledge and similar experience in previous successful programs), as well as for fostering an environment conducive to low staff turnover.

Software Program Managers Network Products and Services

To accomplish Software Best Practices Initiative's objectives, the **Software Program Managers Network (SPMN)** provides specialized products and services, which include the following. *[For more information about the Software Program Manager's Network, contact the customer service center listed in Volume 2, Appendix A or access the SPMN online at the Web address found in Appendix B.]*

Training

Training provides an understanding of essential software acquisition best practices relevant to software practitioners and managers and the techniques for effectively implementing them. The **training mechanisms** available from the Network are:

- Direct satellite broadcast,
- Video and audio tape guide series,
- Intelligent tutoring system, and
- Partnership with the Defense Systems Management College (DSMC).

The software acquisition **core disciplines** and **techniques** for implementing essential best practices include:

- **Software best practices series.**
  - Software risk identification, mitigation, management techniques,
  - Signs of risks turning into problems,
  - Program task planning and scheduling,
  - Planning and baselining techniques,
  - Metrics collection and assessment,
  - Metrics based projection,
  - Earned-value cost, schedule monitoring techniques, and application tips,
  - Earned-value indicators and techniques,
  - Effective software estimation techniques,
  - Software aspects of work breakdown structure
  - Program visibility techniques,
  - Software management metrics,
  - Formal software development inspection techniques,
  - Software complexity analysis techniques,
  - Software test coverage,
  - Software configuration management,
  - Software organizational process improvement,
  - Quality assurance,
  - IV&V: what's effective,
  - Software safety and multi-level security,
  - The role of the independent testing community in software development,
  - Tailoring software specifications and documents,
  - Planning and conducting design and code walkthroughs,
  - Operational test criteria for software maturity, and
  - Other topics.
- **Lessons-learned series.**
  - Software success stories from BSY-2,
  - Software success stories from F-22,
  - Software success stories from F/A-18, and
  - Other lessons-learned.

- **Best practices tips and techniques.**
  - How to eliminate unnecessary and expensive documentation,
  - Risk management realities and lessons-learned, and
  - Other tips and techniques.
- **Technical update series.**
  - Software development strategies: spiral, incremental, and evolutionary;
  - Survey of technology issues for software program managers;
  - I-CASE: productivity, quality, breakthroughs, and cautions; and
  - Rate Monotonic Analysis for real-time systems.
- **Roundtable series.** Candid and unrehearsed discussions by panels of managers and practitioners on current issues, problems, and solutions.
- **Visit with the alchemist series.** The alchemists are industry visionaries, authors, and expert practitioners. Each program provides viewers with the opportunity to call in their questions directly to their software sages.
- **Software contracting series.**
  - Acquisition and contracting strategies, objectives, and techniques,
  - Structuring an RFP, and
  - Effective award-fee structuring, evaluation, and determinations.

Support Services and Products

The Network provides expert help in program assessment and improvement, program simulation, guidebooks, and a means for program managers to share experiences. Support services and products include the following.

- **Alpha Support Team.** This team of highly experienced software management experts provide program managers with independent program assessments. These experts provide a refreshingly new perspective and help identify risks, hidden problems, appropriate metrics, and other paths to consider. The support team's findings and recommendations are private —provided only to the program manager who requested the support assessment; no indication is made of which programs have received or requested this support.
- **Guidebooks.** Based on actual program experiences, these guidebooks provide practitioners and program managers with practical guidance to effectively plan, implement, and monitor their programs. Guidebooks now available or being developed include:
  - **Methods for Managers.** Provides insight into managing, engineering, assurance, and reporting methods. It addresses the

need to develop a consistent program environment; guidance on how to develop, implement, and monitor adherence to policies, standards, and procedures; and tips to follow as program conditions change.

- **Risk Management and Control**. Provides insight into how to plan, organize, execute, and monitor an effective risk management program. Includes the types of risks and indicators of occurrence that concern a government software manager. The need to plan contingencies is also addressed.
- **Estimation and Scheduling**. Provides insight into how to estimate a program and how to schedule activities. It addresses estimation techniques and what to do when expenditures exceed estimates. It also addresses different schedule types, how to develop them, at which schedule levels a government software manager should be concerned, how to monitor schedules, and what to do when they slip.
- **Safety**. Provides insight into the planning, implementing, managing, and process control of large-scale software development and maintenance programs to assure safety requirements are reflected in the operational environment.
- **Configuration Management**. Provides insight into how to plan, organize, execute, and monitor an effective configuration management process. Included are baseline management and control, and configuration management and control of shared program information not formally baselined.
- **Security**. Provides insight into the planning, implementing, management, and control of program security. It also lists what to consider when defining and implementing program security requirements.
- **Product Assurance.** Provides insight into how to plan, organize, execute, and monitor the assurance aspects of a software program. Included is software quality assurance, IV&V, integration and test, reviews and audits, program metrics, and corrective action.
- **Software Program and Organizational Planning**. Provides insight into what types of reports to require; program factors to evaluate; what to look for; and what to do when problems are observed, suspected, or verified.

- **Program manager conferences.** The Network conducts symposia to collectively address specific software acquisition problems, and conducts workshops and conferences to identify and convey software acquisition best practices.
- **Best practices workshops**. An annual workshop of industry, academia, and government software practitioners to review, refine, and recommend software acquisition best practices.

- **Information exchange meetings**. The Network arranges and hosts periodic information exchange meetings for experienced program managers to present and discuss their lessons-learned and management methods. Edited videotapes and audiotapes of these meetings are available to Network members.
- **Focus group**. This group of software practitioners and program managers advises the Network on effective training mechanisms and the sequence in which software disciplines should be taught.
- **Plans and templates library**. Software development plans and similar documents from a successful program can provide a useful "*template*" for others. Library document categories include:
  - Program Plans
  - Software Development Plans
  - Risk Management Plans
  - T&E Plans
  - Source Selection Evaluation Plans
  - Metrics Collection and Analysis: Policies and Practice Plans
  - Top Level Design Plans
  - Configuration Management Plans
- **Tools and literature evaluation**. Provides software practitioners and program managers with features and potential benefit summaries of newly published techniques and methods which enable software managers and practitioners to more efficiently and effectively develop, maintain, and manage large-scale software programs.
- **Lessons-learned Internet database**. Although software acquisition lessons-learned databases or historical chronicles exist at various DoD and government agencies, the Network is identifying these information sources and collecting and loading lessons-learned information into a centrally accessible user-friendly database. This database will be available via Internet World-Wide Web. *[See Volume 2, Appendix B for a list of Web addresses.]* Key lessons from this database collection are conveyed in video/audio tapes and in the Network's *NetFocus* newsletters.
- **PM simulation**. A computer-based, user-friendly method for presenting experiences, challenges, and opportunities derived from actual programs. This simulation familiarizes program managers with management challenges and consequences of various alternative strategies and actions.
- **Technical book digests**. Provides abstracts of selected books on software or other areas as they impact the management, development, and maintenance of large-scale software systems.

SOFTWARE TECHNOLOGY SUPPORT CENTER (STSC)

The **Software Technology Support Center (STSC)** was established in 1987, with a mission to assist USAF organizations in identifying, evaluating, and adopting technologies that improve software product quality, production efficiency and predictability. Since that time, the STSC has provided technical evaluation and consulting services to Air Force and other DoD and federal agencies, including: Air Combat Command, Air Mobility Command, Air Force Materiel Command, Air Force Space Command, Ogden Air Logistics Center, Oklahoma City Air Logistics Center, Warner-Robbins Air Logistics Center, the US Navy, US Special Operations Command, US Strategic Command, and the US Treasury Department.

STSC uses "*technology*" in its broadest sense to include processes, methods, practices, techniques, and tools that enhance human capability. The STSC's focus is on field proven technologies that benefit customers. STSC goes beyond identification and evaluation of technologies by providing hands-on help and guidance in incorporating technologies that increase the organization's value. *[For more information about the STSC, or to request services and publications, contact the STSC Customer Service Center (see Volume 2, Appendix A for the phone number and address) or access the STSC online (see Volume 2, Appendix B for the Web address).]*

Services

The STSC provides five main services to its customers:

- Development and access to **Software Technology On-Line**, a one-stop Web page on the Internet to access meaningful software information. In addition to electronic access to STSC information, it provides pointers to resources across the nation designed to increase customer awareness and enhance understanding. Started in 1989, *Software Technology On-Line* has over 5,000 registered customers.
- **Technical Evaluation Services** provide experienced resources to help identify, pare down, evaluate, and select proven technologies to improve software production. These services are provided on a cost recovery basis and take the form of research, validation, demonstration, evaluation, comparison, analysis, and recommendations. The STSC specializes in Ada, configuration

management, documentation, formal inspections, program management, process definition, re-engineering, requirements engineering, reuse, software design, software estimation, software measurement and metrics, software quality engineering, software testing, and software source selection.

- **Technical Consulting Services** provide experienced engineers and resources to help customers assess, prepare, plan, apply, and effectively use software technologies. These services are provided on a cost recovery basis and take the form of assessments, workshops, counsel, or full-blown programs. The STSC offers experienced resources in the technologies mentioned above.
- Development, publication, and distribution of ***CrossTalk — The Journal of Defense Software Engineering***. This official DoD publication features articles, reports, and opinions from the software community that instruct, inform, and educate its readers. *CrossTalk* is free upon request and has a monthly circulation of 19,000.
- Management and facilitation of the **Software Technology Conference** held annually in Salt Lake City, Utah. This is DoD's premier software event of the year. The conference provides a forum for over 2,800 of the software community's best and brightest to share lessons-learned in acquiring, developing, and supporting software-intensive systems.

THE GUIDELINES, SOFTWARE BEST PRACTICES, AND YOU

To date, these Guidelines represent the most comprehensive compilation of "***software best practices***" available anywhere — Government, academia, or industry. Employ them, and you will be well on your way towards achieving DoD's New Order of Acquisition. Software best practices start with you, your acquisition, and your relationship with your contractor(s). The first step in achieving our National goals is for you to become a world class ***customer***.

BE CAREFUL! **For software, commercial practice is not necessarily *the* best practice. Likewise, some traditional government approaches to software development and management have not been the *"best."* These Guidelines are a first step in identifying the *"best practices"* for software development and management. Use them with authority and a sense of purpose!**

REFERENCES

[ARMSTRONG93] Armstrong, C. Michael, "Re-inventing Government-Industry Relations: Facts of Global Life," speech delivered to the MICOM Advance Planning Briefing for Industry, Huntsville, Alabama, November 9, 1993

[AUGUSTINE95] Augustine, Norm, "Martin Marietta's CEO Speaks to Program Manager," *Program Manager: Journal of the Defense Systems Management College*, March-April 1995

[BROWN95] Brown, Norm, as quoted in "Mandate Is to Ensure Software Will Fly," *Government Computer News*, March 20, 1995

[BUSEY95] Busey, ADM James B., IV, "Battlefield Technologies Muster in Synthetic Arenas," *SIGNAL*, July 1995

[COATES95] Coates, Lt Col R.A., "Demonstrations to be Conducted by the US Marine Corps during Joint Warrior Interoperability Demonstration (JWID)—1995," briefing presented to The Seventh Software Technology Conference, Salt Lake City, Utah, April 12, 1995

[COHEN95] Cohen, Senator William S., as quoted by Tim Minahan, "US Can Win with Best IT, Gingrich Says," *Government Computer News*, February 20, 1995

[DANE90] Dane, Abe, "Black Jet," *Popular Mechanics*, July 1990

[DISA95] Defense Information Systems Agency, "Technical Architecture Framework for Information Management (TAFIM)," briefing presented by Virginia L. Conway to the DoD Software Reuse Initiative Technical Interchange Meeting for Product-Line Architectures, June 19, 1995

[DRELICHARZ94] Drelicharz, Joseph A., "Highlights of the Federal Acquisition Streamlining Act of 1994," *Program Manager: Journal of the Defense Systems Management College*, March-April 1995

[DSMC90] Caro, Lt Col Isreal, et al., Mission Critical Computer Resources Management Guide, Defense Systems Management College, Fort Belvoir, Virginia, 1990

[ECSSP91] *United States Air Force (USAF) Embedded Computer Systems (ECS) Standardization Plan*, SAF/AQK, The Pentagon, Washington, DC, October 15, 1991

[EDMONDS93] Edmonds, Lt Gen Albert J., as quoted by Joyce Endoso, "Ada Gets Credit for F-22's Software Success," *Government Computer News*, April 26, 1993

[ENGELLAND90] Engelland, J.D., "Integrated Avionics Systems: Managing the System and the Software," briefing present by General Dynamics to Boldstroke, Maxwell AFB, November 8, 1990

[FAIN92] Fain, Lt Gen Jim, as quoted by Lt Gen Robert H. Ludwig, "The Role of Technology in Modern Warfare," briefing presented to the Software Technology Conference, April 14, 1992

[FORNELL92] Fornell, Lt Gen Gordon E., cover letter to draft report, "Process for Acquiring Software Architecture," July 10, 1992

[GAO95] General Accounting Office, Testimony before the Committee on the Budget, House of Representatives by Frank C. Conahan, *Defense Programs and Spending: Need for Reforms*, GAO/T-NSIAD-95-149, April 27, 1995

[GARCIA94] Garcia, Andrea, "Dr. Kaminski Delivers Keynote Address: Transforming the Way We Buy Goods and Services," *Program Manager: Journal of the Defense Systems Management College*, March-April 1995

[GCSS95] World Wide Web, http://www.ssc.af.mil/XO/XON2/vision.htm, March 1995

[GUENTHER95] Guenther, LGEN Otto, as quoted by Bob Brewin, "Services Struggle with On-going 'Software Crisis,'" *Government Computer Week*, April 24, 1995

[HAYES93] Hayes, Patrick J., "Is Artificial Intelligence Real?: Absolutely—and Vital for Riding the Rising Tide of Information, *Washington Technology*, September 9, 1993

[HOROWITZ95] Horowitz, Barry M., personal correspondence to Lloyd K. Mosemann, II, September 8, 1995

[HUEY91] Huey, John and Nancy J. Perry, "The Future of Arms," *Fortune*, February 25, 1991

[JCS72] The Joint Chiefs of Staff, Department of Defense Dictionary of Military and Associated Terms, Government Printing Office, Washington, DC, 1972

[JONES94] Jones, Jennifer, "DoD Moves Toward 'Best Practices' Panel to Steer Toward Successful Approaches to Projects," *Federal Computer Week*, September 26, 1994

[KAMINSKI94] Kaminski, Paul, as quoted by Andrea Garcia, "Dr. Kaminski Delivers Keynote Address: Transforming the Way We Buy Goods and Services," *Program Manager: Journal of the Defense Systems Management College*, January-February 1995

[LUDWIG92] Ludwig, Lt Gen Robert H., "The Role of Technology in Modern Warfare," briefing presented to the Software Technology Conference, April 14, 1992

[OBENZA94] Obenza, Ray, "Guaranteeing Real-Time Performance Using RMA," *Embedded Systems Programming*, May 1994

[OSA92] "Tri-Service Open System Architecture Working Group," briefing, 1992

[OWENS95] Owens, ADM William A., as quoted by Howard Banks, "Fewer Ships, More Microchips," *Forbes*, June 19, 1995

[PAIGE93] Paige, Emmett, Jr., as quoted by Thomas R. Temin, "Top Defense IT Exec Puts Some of His Cards on the Table," *Government Computer News*, July 19, 1993

[PARKER89] Parker, Sybil P., ed., McGraw-Hill Dictionary of Scientific and Technical Terms, Fourth Edition, McGraw-Hill Book Company, New York, 1989

[PETERSEN92] Petersen, Gary, "The Spectrum of STSC Support," *CrossTalk*, Issue 32, March 1992

[POWELL92] Powell, GEN Colin L., "Information-Age Warriors," *Byte*, July 1992

[PRESTON95] Preston, Colleen, as quoted by Collie J. Johnson, "Secretary Preston Underscores Dramatic Changes in DoD's Acquisition Arena," *Program Manager: Journal of the Defense Systems Management College*, March-April 1995

[ROOS95] Roos, John G., "New and Newer Submarines: As Seawolf Prepares to Prowl the Depths, A Likely Successor Already Roams—In Cyberspace," *Armed Forces Journal INTERNATIONAL*, July 1995

[SMITH94] Smith, Bruce A., "Downsizing to Deepen As Backlogs Shrink," *Aviation Week & Space Technology*, March 14, 1994

[TOFFLER93] Toffer, Alvin and Heidi Toffler, War and Anti-War: Survival at the Dawn of the 21st Century, Little, Brown, and Company, Boston, 1993

[WENTZ92] Wentz, Larry K., "Communications Support for the High Technology Battlefield," The First Information War, AFCEA International Press, Fairfax, Virginia, October 1992

[WHITE80] White, Eston T., Defense Organization and Management, National Defense University, Washington, D.C., 1980

[ZRAKET92] Zraket, Charles E., "Software: Productivity Puzzles, Policy Changes," John A. Alic, ed., Beyond Spinoff: Military and Commercial Technologies in a Changing World, Harvard Business School Press, Boston, Massachusetts, 1992

Blank page.

CHAPTER 2
Addendum A

MIL-STD-498: What's New and Some Real Lessons-Learned

Paul A Szulewski
David S. Maibor

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 2
Addendum B

Adopting MIL-STD-498: The Stepping-stone to the US Commercial Standard

Reed Sorensen
Software Technology Support Center

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

Blank page.

CHAPTER 3

System Life Cycle and Methodologies

CHAPTER OVERVIEW

When accused of making snap decisions, General George S. Patton, Jr. (known as Old Blood and Guts) firmly proclaimed,

> *I've been studying the art of war for forty-odd years. When a surgeon decides in the course of an operation to change its objective...he is not making a snap decision but one based on knowledge, experience, and training. So am I.* [PATTON47]

The life cycle process was developed so managers of major defense acquisition programs are not forced into making snap decisions. Rather, a structured process, replete with controls, reviews, and key decision points, provides the basis for sound-decision making based on knowledge, experience, and training. In this chapter you will learn that the life cycle process is a logical flow of activity representing an orderly progression from the identification of a mission need to final operational deployment and support.

As a program manager, you must be prepared to develop a tailored management approach that will achieve an operational capability with the most effective use of resources and time available. Choosing the right life cycle management methodology for your program depends on the nature of its operational environment, stability of requirements and technology, your software domain, the methods and tools used, and the controls and deliverables required. If appropriate, life cycle phases can be combined or omitted. Some programs [especially in the case of MIS where technology is well-developed, purchased as commercial-off-the-shelf (COTS), or government-off-the-shelf (GOTS)] may enter the life cycle midstream. Each life cycle phase is designed to develop a level of confidence in the solution(s) offered and to reduce the risks involved in making a decision to proceed to the next phase. Outputs of each phase constitute a definitive, documented baseline for entry into subsequent phases.

Blank page.

CHAPTER 3

System Life Cycle and Methodologies

LIFE CYCLE PROCESS RAISES CURTAIN ON DECISION MAKING

Prior to the early 1970s, defense acquisition decision making was often similar to how military historian and analyst, Brigadier General Samuel L.A. Marshall, described decision making during combat:

> *In war, much of what is most pertinent lies behind a drawn curtain. The officer is therefore badly advised who would believe that a hunch is without value, or that there is something unmilitary about the simple decision to take some positive action, even though he is working in the dark.* [MARSHALL51]

Making decisions behind a drawn curtain might well be a necessity of battle, but it can prove costly in terms of quality, safety, and performance when procuring weapons to go to war. **Office of Management and Budget Circular A-109** was published in 1976 to throw light on the acquisition process and to define a decision mechanism based on quantitative assessments, reviews, and audits of the life cycle process. It established policies, methods, procedures, a life cycle, and milestone decision process to increase effectiveness in decision making for all major system acquisitions. For weapon systems, C2, and MIS programs, **milestone decision points** mark the completion of one phase of the life cycle and entry into the next. Peer reviews, completion of measured process activities, the production of defined work products, audits, and other evaluation procedures throughout each phase support exit and entry criteria for milestone decisions.

SYSTEM LIFE CYCLE

To understand the system life cycle and its acquisition phases, it is important to realize how they relate to the systems engineering and software engineering processes *[discussed in Chapter 4, Engineering Software-Intensive Systems]*. A **system** is defined as "*an integrated composite of people, products, and processes that provide a capability to satisfy a stated need or objective.*" A **subsystem** (e.g., the software) is "*a grouping of items satisfying a logical group of functions within a particular system.*" [MIL-STD-499B] Software must always interface with the other elements that make up the total (weapon or MIS) system. For example, MIS systems may have several hundred interfaces with other essentially independent MIS and C2 systems. Embedded flight control software must reflect the control surfaces and atmospheric buffeting affected by an aircraft's physical design. In either case, this larger ***system-of-systems*** context must be taken into account. Given these relationships and interdependencies, it is vital to have a ***systems-view***, as expressed by Field Marshall Viscount Montgomery.

> *It is absolutely vital that a senior commander should keep himself from becoming immersed in details, and I always did so...In battle a commander has got to think how he will defeat the enemy. If he gets involved in details he cannot do this since he will lose sight of the essentials which really matter; he will then be led off on side issues which will have little influence on the battle, and he will fail to be that solid rock on which his staff can lean. Details are their province. No commander whose daily life is spent in the consideration of details...can make sound plan of battle on a high level or conduct large-scale operations efficiently.* [MONTGOMERY58]

The elements making up a system are illustrated in Figure 3-1. Hardware, software, and documentation are explained in Chapter 2, *DoD Software Acquisition Environment*. Two additional elements included in a system, **database** and **procedures**, are defined as:

- A **database** is a large, organized collection of information accessed by software that is an integral part of the system's function.
- **Procedures** are the steps defining the specific use of each system element or the procedural context in which the system resides. [PRESSMAN92]

CHAPTER 3 System Life Cycle and Methodologies

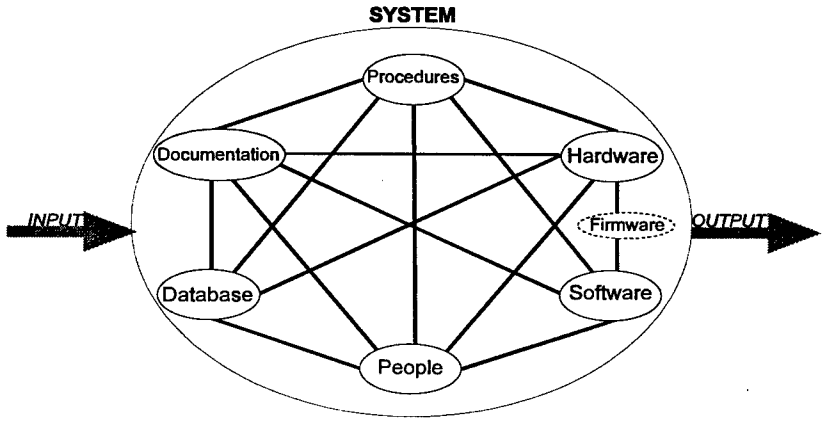


Figure 3-1 Software-Intensive System

The life cycle phases for weapon systems and MISs are quite similar. Aside from assembling/integrating components, the hardware [computer(s) and operating system(s)] is not usually specifically manufactured for MISs. ***However, never lose sight of the fact that hardware and software development are intimately related.*** Although they are often developed independently, concurrently, and/or purchased separately, software is almost always on the critical path of a software-intensive system. Therefore, it is important to consider early how the software is to work within the system and what software interfaces are necessary to achieve the benefits of cohesive, interoperable components. Proper **integration** of the hardware and software can be assured through carefully identified **interface requirements**, prudently planned demonstrations, and system/subsystem tests and evaluations. Such techniques improve accuracy, currency, and quality of information.

NOTE: The international standard ISO/IEC 12207, *Information Technology — Life Cycle Processes*, was published in 1995. This standard establishes a common framework for software life cycle processes. It contains processes, activities, and tasks that are to be applied during the acquisition of a system containing software, a stand alone software product and service, and during the supply, development, operation, and maintenance of software products. This standard will be used to implement and replace MIL-STD-498.

Life Cycle Phases, Milestone Decisions, and Activities

Air Force Instruction (AFI) 10-601, *Mission Needs and Operational Requirements Guidance and Procedures,* describes the earliest activities of the system life cycle. The services are continuously performing and updating **Mission Area Assessments (MAAs)** to identify mission needs using a *strategy-to-task* process which links the need for military capabilities to the strategy provided by the Chairman of the Joint Chiefs of Staff (CJCS). The ability to accomplish the *tasks* from the *strategy-to-task* process, using current and programmed systems, is then evaluated in a **Mission Need Analysis (MNA).** This process is called "***task-to-need.***" The products of MAAs and MNAs are used to construct a **Mission Area Plan (MAP).** The MAP is a strategic planning document that covers approximately 25 years and records the proposed plan for correcting mission capability deficiencies. It expresses nonmateriel solutions, including changes in force structure, system modifications or upgrades, science and technology applications, and new acquisition programs.

Upgrade, modification, and new acquisition programs are established when nonmateriel solutions do not provide adequate fulfillment of an identified task deficiency. The needed capability is documented in a **Mission Need Statement (MNS).** The MNS is a brief statement (no more than 5 typed pages) identifying and documenting mission deficiencies requiring materiel and/or software solutions:

- To define an operational need,
- To officially validate an operational need, and
- To furnish implementation and support to OT&E activities. [AFI 10-601]

As capability(ies) become better defined, specific requirements levied on the new or modified system are stated in the **Operational Requirements Document (ORD).** During the pre-Milestone 0 phase, the need for the acquisition program is studied and documented. The **Concept Studies Approval** (Milestone 0) decision is made when the **Milestone Decision Authority (MDA)** determines that the mission need:

- Is based on a validated projected threat,
- Cannot be satisfied by a nonmateriel solution, and
- Is sufficiently important to warrant the funding of study efforts to explore and define alternative concepts to satisfying the need.

Concept Exploration and Definition (Phase 0). Following a successful **Milestone 0** decision, this phase involves a series of studies to identify the best possible solutions to the mission need in terms of cost, risk, schedule, and readiness objectives. These various concepts are often explored through competitive, parallel, short-term contracts focused on defining and evaluating the feasibility of alternative concepts. They provide the basis for assessing the relative merits of the concepts at the **Milestone I** decision point. Life cycle cost estimates, logistic support analysis, and producibility engineering assessments are prepared along with the **Operational Requirements Document (ORD)**. The ORD is solution-oriented and defines the tradeoff studies conducted during this phase. It becomes the basis for program direction, program baselines, the **Integrated Master Plan (IMP)** and **Integrated Master Schedule (IMS)**, and subsequent development of the **Test and Evaluation Master Plan (TEMP)**. *[A list of recommended items to include in a draft ORD is found in Volume 2, Appendix N.]* A product of this phase is the selection of a proposed **Acquisition Strategy** *[discussed in Chapter 12, Planning for Success]*. Demonstration program(s) are designed, coded, tested, and implemented to provide basic, or elementary, capabilities across the full range of requirements.

Demonstration and Validation (Dem/Val) (Phase I). Following a successful ***Milestone I*** decision to proceed, Dem/Val (Phase I) consists of activities dependent on the choice of a life cycle management methodology. When warranted, multiple design approaches and parallel technologies are pursued within the system concept. One of the most important aspects of this phase is the early integration of supportability considerations into the system design concept. As you will learn throughout these Guidelines, ***addressing supportability early in the life cycle precludes later costly redesign efforts***. This is an important consideration, as the decisions made during this phase impact approximately **60%** of the total life cycle cost (as illustrated on Figure 3-2 below), which will be spent later during the Operations and Support phase (as illustrated on Figure 3-3 below).

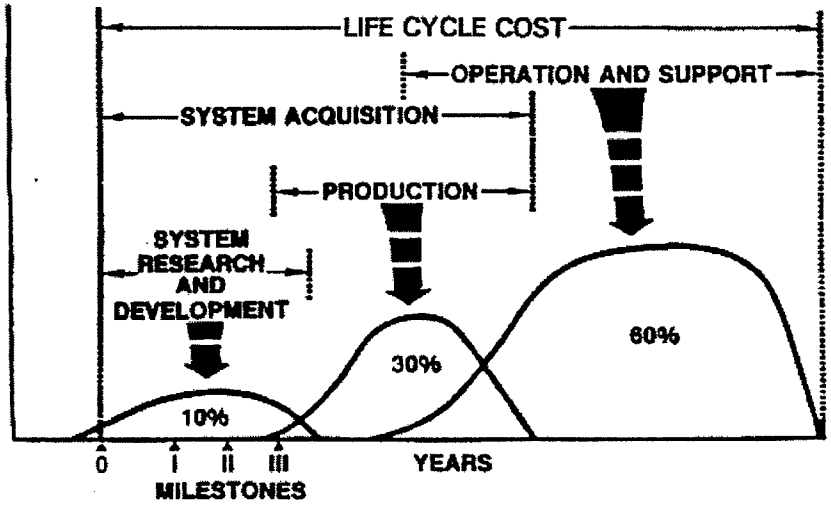


Figure 3-2 Nominal Cost Distribution of a Typical DoD Program [DSMC90]

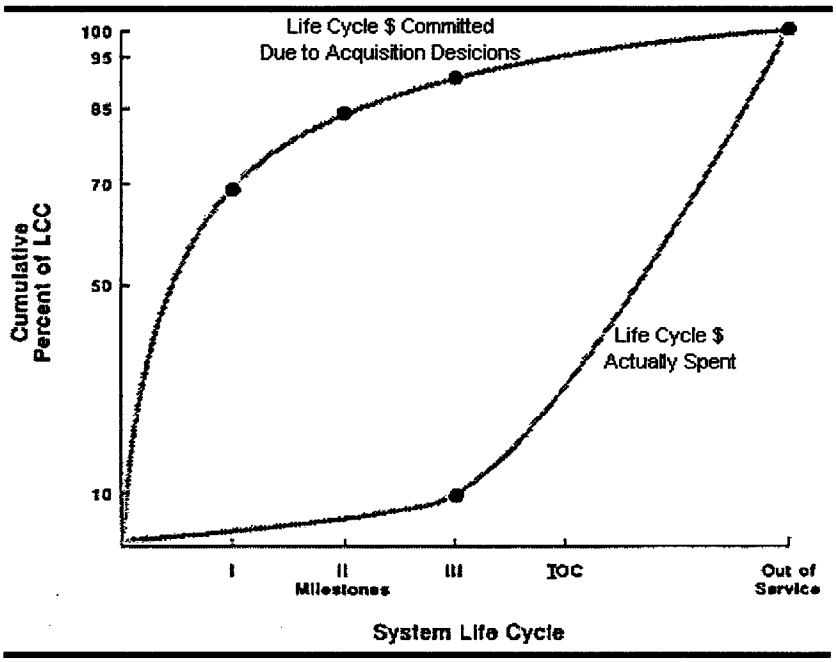


Figure 3-3 Effect of Early Decisions on Life Cycle Cost [DSMC90]

As another risk and cost reduction measure, prototyping, testing, and ***early user involvement*** in operational assessments of critical components cannot be overemphasized. Cost drivers and alternatives are identified and analyzed. As a function of risk, the costs of alternative design approach(es) must be evaluated against increases in performance capabilities. *[See Chapter 6, Risk Management, for further discussion of cost risk.]* The ORD, TEMP, and Acquisition Strategy are updated to reflect the work performed in Phase I. The outcome of this phase is the **Milestone II** decision, where the affordability of the program is assessed and a decision is made on whether the activities of this phase warrant continuation to the next phase. A **Development Baseline** is established reflecting cost, schedule, and performance requirements.

Engineering and Manufacturing Development (Phase II) **(Development** for MIS). Following a successful Milestone II decision, this phase may be repeated if the life cycle methods are incremental or evolutionary *[discussed below]*. Effective risk management *[discussed in Chapter 6, Risk Management]* is especially critical. Resources are only committed commensurate with the reduction and closure of identified risk items. Requirements are placed under configuration control. Configuration control is also implemented for both the design and development process. System-specific performance capabilities are developed in coordination with user approval. Assessments of performance, schedule, and cost are made throughout this phase and compared with the TEMP. Acquisition Strategy are updated as appropriate. Developmental test and evaluation (DT&E) (test to development specification) is performed. DT&E also supports, and provides data for, operational assessment prior to the operational test and evaluation (OT&E) (test to operational requirements) which is also performed during EMD. The outcome of the EMD phase is the **Milestone III** decision, when it is determined whether the activities of this phase warrant continuation to the next phase. A **Production Baseline** is established that reflects cost, schedule, and performance assessment requirements for the next phase. According to **Watts Humphrey**,

> *As long as programmers are writing code, they are making design decisions, just at a more detailed level. Many of these details will impact the usability and performance of the system, just not at a high enough level for the people who wrote the requirements to be aware of them. The field users of such systems, however, will*

> *almost always find that systems developed blindly from requirements documents are inconvenient and unwieldy in operational use. Truly superior usability can only be obtained when the developers have an in-depth knowledge of actual field conditions. While suppliers should start from official requirements, these must be recognized as a starting point and that much more detailed knowledge is required before the system can actually be built. The key is to make the supplier responsible for devising, defining, and using a process that uncovers true operational requirements.* [HUMPHREY95]

Production and Deployment (Phase III). Following a successful Milestone III decision, system performance and quality are monitored during this phase by follow-on OT&E (FOT&E). Cost, schedule, and performance are periodically reviewed and compared to the Production Baseline. User feedback and the results of field experience, to include operational readiness rates, are continuously monitored. Support plans are implemented to ensure sufficient support resources are acquired and deployed with the system.

Operations and Support (Phase IV). This phase overlaps with Phase III and begins after initial systems, increments, or capabilities have been fielded. This phase is marked by either the declaration of an operational capability or the transition of management responsibility from the developer to the maintainer. It continues until the system is retired from the inventory or a **Milestone IV** decision is made to commit to a major upgrade or modification (which causes the program to re-enter Phase I, II, or III, as appropriate). Quality, safety, performance, and technological obsolescence are corrected as identified. Modifications and updates are undertaken to extend the system's useful life with care taken to minimize proliferation of system configurations. Post deployment supportability/readiness reviews are periodically conducted to resolve operational and supportability issues.

LIFE CYCLE MANAGEMENT METHODOLOGIES

A **methodology** refers to the standards and procedures that affect the planning, analysis, design, development, implementation, operation, support, and disposal of a software-intensive system. Thus, we use the term ***software life cycle management methodology***, rather than software development methodology, to avoid a perception that the methodology only focuses on the design and build stages. Developed from historical program experience, methodologies provide insight into the use of candidate solutions based on program character, acceptable level of risk, and program constraints. Methodologies also present a conceptualization of the life cycle process that can be used as a communication tool among all system stakeholders. Specifically, life cycle management methodologies aid in determining the sequence of major life cycle activities, provide a better understanding of the processes required for each activity, and serve as a starting point from which management decisions can be made. One thing to remember is that software development methodologies used for weapon systems must integrate with, and be consistent with, the weapon system and systems engineering development methodologies used for the total program.

Software life cycle management methodologies include evolutionary, incremental, spiral, waterfall, or any other method chosen for its applicability to your developmental or support environment. [PASSMORE94] A fast-track life cycle methodology speeds up (or bypasses) one or more of the life cycle phases or development processes. An organization can use an existing methodology or develop its own. The focus, names of components, and division of activities vary among methodologies. A life cycle methodology should be chosen based on the nature of your program, software domain, the methods and tools used, and the controls and deliverables required. [PRESSMAN92] Most life cycle management methodologies include at least the following:

- **Phases.** The methodology should divide the life cycle into phases, noting which activities fall in each, and include a process for determining when each system component can move to the next phase.

- **Milestones.** The methodology should define the milestones in each phase. Milestones should be event-driven, rather than schedule of cost driven. Each milestone should specify appropriate deliverables (e.g., a written report, briefing, test result, portion of code, or analysis and design data). The methodology should also include criteria for when the program office approves completion of one phase and movement into the next.
- **Content of deliverables.** The methodology should define, either by topic or outline, what each milestone deliverable should include.
- **Evaluation criteria for deliverables.** The methodology should define what criteria a deliverable must meet for formal acceptance by the Government as having satisfied the milestone. *[These are also defined as **exit criteria** for completion of the phase and passage to the next milestone].* Both the methodology and criteria should be specified in the **Software Development Plan (SDP)** *[discussed in Chapter 14, Managing Software Development. See Chapter 15, Managing Process Improvement, for a discussion on "exit criteria" in respect to earned-value analysis.]*

During software development, errors/defects are discovered, opportunities are revealed, changes are superimposed, and even changes are changed. Unless carefully controlled, the ensuing complexity makes the software evolution error-prone, time consuming, and expensive. The use of life cycle management methodologies has proven to be extremely effective in controlling change and in managing the complexity of the development process. However, for any life cycle methodology to be effective, it must be customized to specific program goals. Therefore, your selected methodology must be adapted and evolved, the same as the technical activities it ties together. Understanding your software process and making tradeoffs between incorporating and deleting life cycle components is crucially important for producing high quality software, on time, within budget.

As useful as they are, life cycle methodologies have their limitations in that they can hide important process detail crucial to program success. In themselves, life cycle management methods are often too abstract to convey the details of architecture, concept of operations, process steps, data flows, development activities, engineering roles, and program constraints. CASE tools *[discussed in Chapter 10, Software Tools]* can aid many facets of the life cycle process, such as data modeling and normalization, graphical support of design

modeling, and code testing. They also support program management, planning, estimation and control, as well as configuration management. [PASSMORE94] But remember, CASE tools are just that — ***tools***. A CASE tool will not tell you what software-intensive system to build, what the system must do, or how it should be designed. This process must evolve from user needs and reflect improvements in development methods, techniques, standards, and available software engineering technology.

Choosing an appropriate life cycle methodology is not always an easy task. All those presented here have unique advantages and limitations that must be considered. Current guidance for MIS development is that either incremental or evolutionary methods *[particularly the Ada Spiral Model, discussed below]* constitute more effective risk management and provide earlier satisfaction of user requirements. These approaches are also recommended for weapon systems, when appropriate. *[See Chapter 10, Software Tools, for guidance on selecting an appropriate CASE tool that models your selected life cycle methodology.]*

Evolutionary

The **evolutionary life cycle** is an alternative strategy for systems where future requirement refinements are anticipated or where there is a technical risk or opportunity that discourages the immediate implementation of a required capability. The evolutionary life cycle method is a strategy in which a core capability is fielded, the system design has a modular structure, and provisions are made for upgrades and changes as requirements are refined. Figure 3-4 (below) illustrates Pressman's interpretation of the evolutionary model where a first generation spiral evolves into a second generation extended spiral. [PRESSMAN93]

An evolutionary life cycle method is well-suited for high technology software-intensive systems where requirements beyond the core capability can generally, but not specifically, be identified. This is usually the case with software-dominated decision support systems that are highly interactive with complex human-machine interfaces. Evolutionary programs are conducted within the context of a ***plan*** for progression towards an ultimate capability. This strategy also requires the development of increments of software demonstrable to the user, who is involved throughout the entire development process, as illustrated in Figure 3-5 (below). An evolutionary methodology can

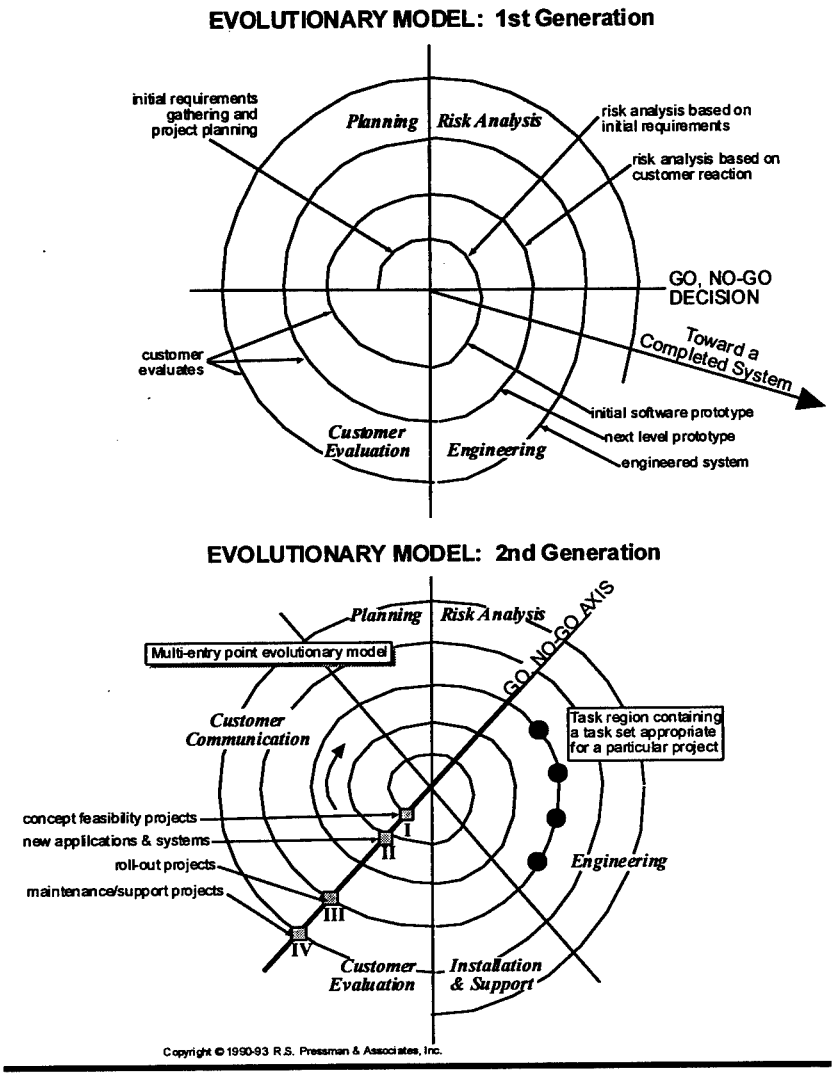


Figure 3-4 Evolutionary Life Cycle Generations [PRESSMAN93]

be employed on all types of acquisitions. However, ***it is mostly used on medium to high-risk programs.*** While the final version of the system often takes more time and effort to develop than other efforts, this additional effort and time is offset by delivery of a better quality product with reduced maintenance cost. According to **Humphrey**,

> *There is a basic principle of most systems that involve more than minor evolutionary change: the system will*

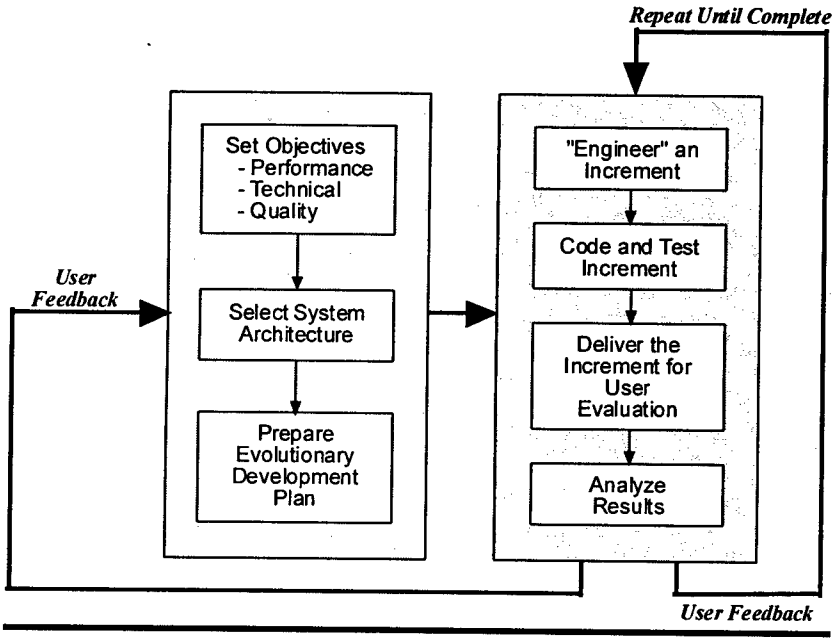


Figure 3-5 User Involvement in the Evolutionary Method

change the operational environment. Since the users can only think in terms of the environment they know, the requirements for such systems are always stated in the current environment's terms. These requirements are thus necessarily incomplete, inaccurate, and misleading. The challenge for the system developer is to devise a development process that will discover, define, and develop to real requirements. This can only be done with intimate user involvement, and often with periodic prototype or early version field tests. Such processes always appear to take longer but invariably end up with a better system much sooner than with any other strategy. [HUMPHREY95]

Incremental

The **incremental life cycle management method** involves developing a software-intensive product in a series of increments of increasing functional capability. Benefits of the incremental method are:

- Risk is spread across several smaller increments instead of concentrating in one large development;
- Requirements are stabilized (through user buy-in) during the production of a given increment by deferring nonessential changes until later increments; and
- Understanding of the requirements for later increments becomes clearer based on the user's ability to gain a working knowledge of earlier increments.

Figure 3-6 illustrates the incremental life cycle method. It allows the user to employ ***part of the product*** and is characterized by a *build-a-little, test-a-little* approach to deliver an initial functional subset of the final capability. This subset is subsequently upgraded or augmented until the total scope of the stated user requirement is satisfied. The number, size, and phasing of incremental builds leading to program completion are defined in consultation with the user. ***An incremental methodology is most appropriate for low to medium-risk programs***, when user requirements can be *fully defined*, or assessment of other considerations (e.g., risks, funding, schedule, size of program, early realization of benefits) indicate that a phased approach is the most prudent.

Figure 3-7 (below) provides an example of how the incremental method might be related to the ***MAISRC*** process [**System Program Director (SPD)** or **Designated Acquisition Commander (DAC)/ Program Executive Officer (PEO)**]. Allowing the user to employ the partially completed product before the entire product is integrated and tested also promotes early discovery of problems and facilitates corrections.

Be aware, while the program as a whole may be large enough to merit MAISRC (or DAB) oversight, incremental development and fielding decisions only address small subsets of the system. For example, a $240 million program with MAISRC oversight may have a Milestone II decision impacting $15 million. It may be inappropriate for the MDA to make a $15 million development milestone decision, although

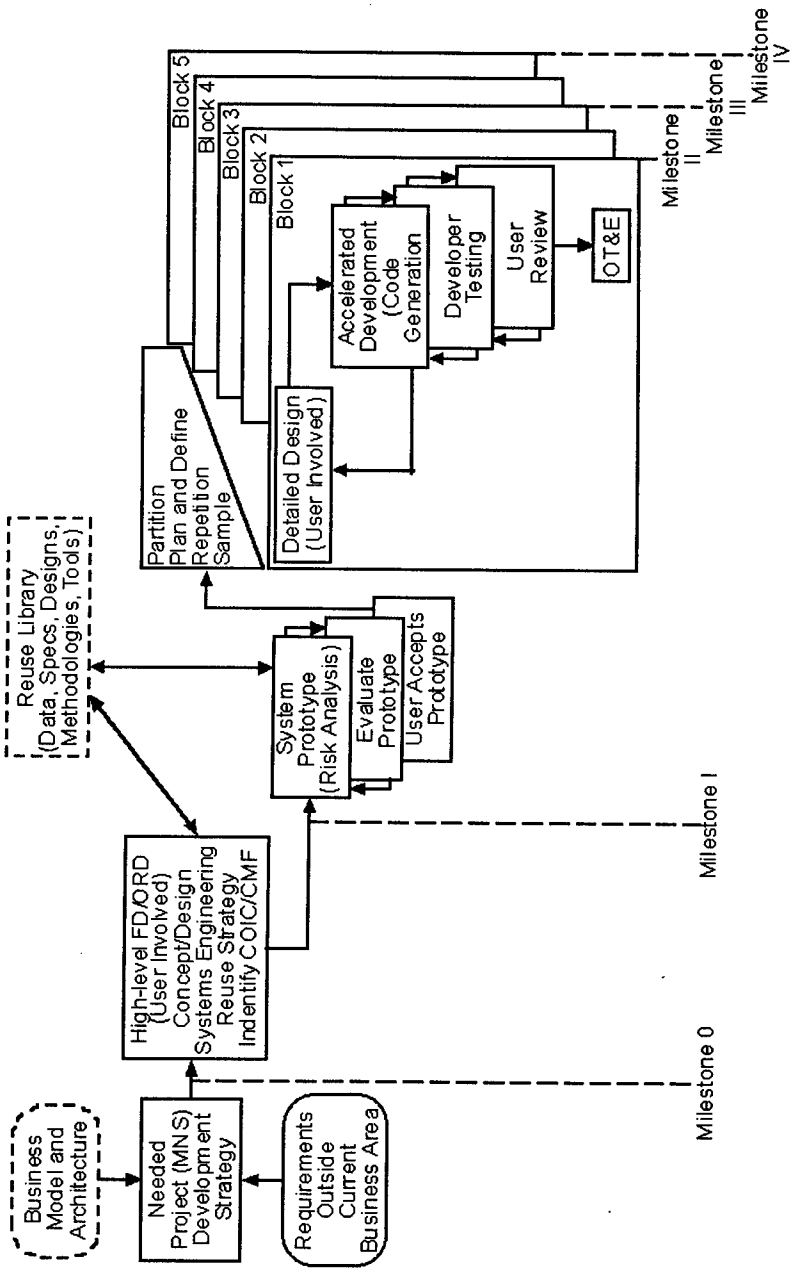


Figure 3-6 Example Incremental Life Cycle Method

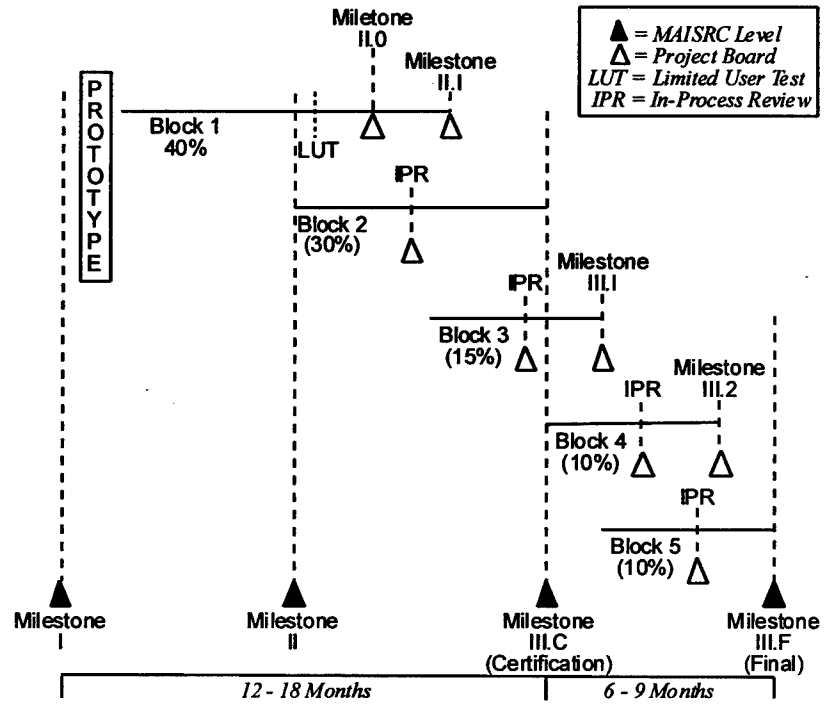


Figure 3-7 Example Incremental Method/MAISRC/Project Board Milestones/Reviews

the decision to develop the subset of system functionality is, in fact, a Milestone II decision (for a small effort). DoDI 5000.2 and DoDI 8120 describe the elements required for a development decision (adequate cost estimate and funds, firm documentation of operational requirements, mature acquisition strategy, and other elements).

Due to their nature, the evolutionary/incremental acquisitions often encounter complications. Questions arise because each incremental build block provides but a small part of the capability of the system to be acquired. In addition to normal development decision criteria, additional questions must be answered, which include:

- Is the decision to develop this functionality for this amount of money a good idea?
- Is this the time to address the functionality question (user priorities, dictates of the evolution itself)?
- Is this a reasonable price for the functionality being added (or are we *goldplating* one functional area before developing all required capabilities)?

- Will we run out of money before we complete the required system?

NOTE: Critical to evolutionary or incremental methods is a sound *architecture* which permits the addition of capability, core enlargement, or added increments.

Spiral Method

Spiral method (implemented by the Spiral Model), developed by **Barry Boehm**, provides a risk reducing approach to the software life cycle. As illustrated in Figure 3-8, in the Spiral Model the radial distance is a measure of effort expended, while the angular distance represents progress. It combines basic waterfall *[discussed next]* building block and evolutionary/incremental prototype approaches to software development. The building block activities of architectural (preliminary) design, Preliminary Design Review (PDR), detailed design, Critical Design Review (CDR), code, unit test, integration and

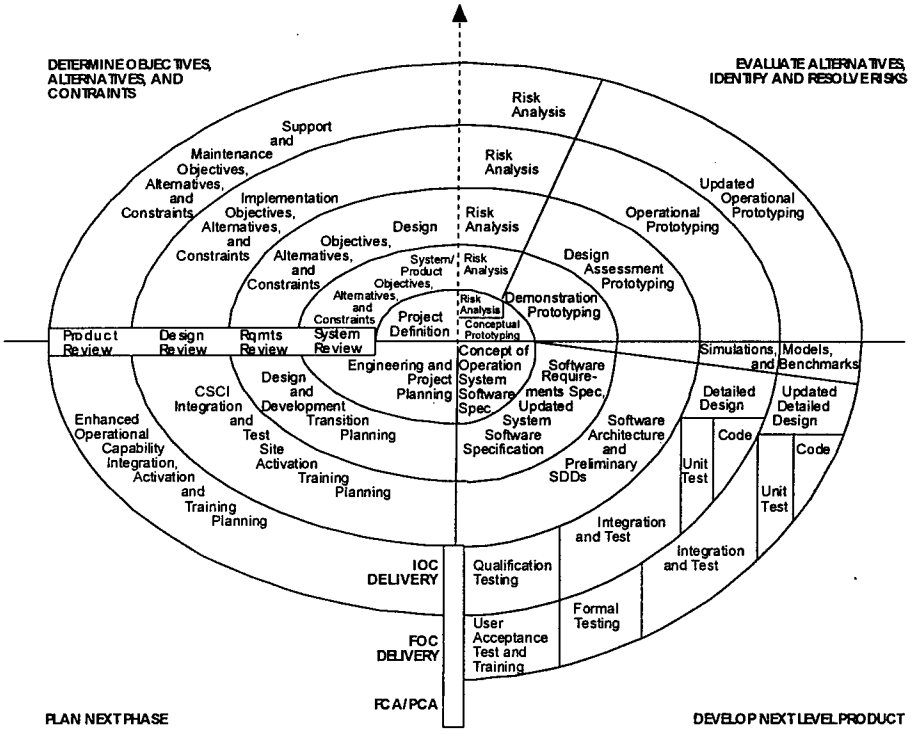


Figure 3-8 Spiral Model

test, and qualification test are sequentially followed to deliver an ***initial operational capability (IOC)***. After IOC, the product is reviewed to determine how its operational capability can be enhanced. Support and maintenance issues are revisited through risk analysis. The product is updated and an operational prototype(s) demonstrated and validated. *[Prototyping is discussed in Chapter 14, Managing Software Development.]* The system then goes through an updated waterfall development process with final delivery of a ***full operational capability (FOC)*** product.

Addressed more thoroughly than with other strategies, the spiral method emphasizes the evaluation of alternatives and risk assessment. A review at the end of each phase ensures commitment to the next phase, or if necessary, identifies the need to rework a phase. The advantages of the spiral model are its emphasis on procedures, such as risk analysis, and its adaptability to different life cycle approaches. If the spiral method is employed with demonstrations, baselining, and configuration management, you can get continuous user buy-in and a disciplined process. [BOEHM88]

Ada Spiral Model Environment

The **Ada Spiral Model Environment** is an adaptation of Boehm's spiral model, as illustrated in Figure 3-8. It constitutes a development environment that combines a model and tool environment. It uses Ada as a life cycle language and offers the ability to have continual *"touch-and-feel"* of the software product (as opposed to paper reports and descriptions). It is a demonstration-based process that employs a top-down incremental approach, resulting in early and continuous design and implementation validation. The advantage of this environment is that not only does it build from the top down, but because Ada supports partial implementation, the structure is automated, real, easily evolved, where each level of development can be demonstrated. Each build and subsequent demonstration validates the process and structure. Each level may serve as a formal baseline that can be controlled. [WOODWARD89]

Choosing Among Evolutionary, Incremental, and Spiral Models

Are you asking, *"What are the differences between the evolutionary, incremental, and spiral models?"* Actually, most programs use a combination of all three. The spiral model is an overlay of either incremental or evolutionary with the addition of risk management. In the past, the spiral model has been difficult to implement purely in the DoD procurement environment because predefined deliverables and schedules do not easily accommodate repeating phases, changing deliverables, or discarding requirements without difficult contract modifications. In a commercial environment, it could be described as *market-driven*, where *time-to-market* and competitive forces determine when a product must be delivered and what features are included (features may change rapidly in light of competitor releases). In reality, all software evolves — commercial products are always evolving, and in DoD the process is called support or maintenance. Key to identifying which system components should follow which development method is driven by such factors as: (1) time to market/ release; (2) understanding requirements; (3) technology obsolescence; (4) priority of user needs; (5) expected life of system; (6) size (magnitude) of effort; (7) complexity; (8) parallel hardware development; and (9) interfaces to existing and unknown (to be developed) systems. [QUANN95] Of course, the order of importance and weighting of each factor varies from program to program and between commercial and military applications.

Waterfall Model

The **waterfall model** was first identified in 1970 as a formal alternative to the *code-and-fix* software development method prevalent at the time. [ROYCE70] The waterfall model was the first to formalize a framework for software development phases, and placed emphasis on upfront requirements and design activities and on producing documentation during early phases. The major drawback to this model is its inherent sequential nature — any attempt to go back two or more phases to correct a problem or deficiency would result in major increases in cost and schedule.

While the waterfall model (also referred to as "*Grand Design*") provided an early structured method for the software life cycle, ***it is not suited for modern development techniques such as prototyping***

and automatic code generation. *[See General Services Administration's, Alternatives to Grand Design for Systems Modernization, for a discussion on when the waterfall is* ***NOT*** *appropriate.]* [GSA91] In the traditional waterfall model, each stage is a prerequisite for succeeding activities, making this method a risky choice for unprecedented systems because it inhibits flexibility. With a single pass through the process, integration problems usually surface too late. Also, a completed product is not available until the end of the process, discouraging user involvement. Taking these factors into account, ***the other life cycle methods discussed in this chapter are recommended instead!***

NOTE: In general, the waterfall method itself is NOT recommended for major software-intensive acquisition programs! If it must be used (due to integrating into the weapon system's overall system engineering methodology) then software management and engineering techniques described throughout this book must also be used to reduce program risk.

Fast Track

Although the focus of these Guidelines is on ***"major"*** software-intensive systems, a distinction between major and non-major programs should be understood. Software-intensive programs not considered major acquisitions and using a fast-track life cycle methodology require greater tailoring of software development tasks. This may be based on a time criticality arising from a variety of reasons, such as a national threat. Although these programs are less formal and on a shortened life cycle to benefit the Government, the primary focus may not be on a time critical schedule (e.g., *proof-of-concept* programs). ***Fast track or abbreviated software-intensive programs always assume short maintenance phases where system support is performed by the development contractor.***

Although process tailoring is necessary to meet shortened life cycle requirements, some of the normal acquisition and development steps are maintained while reducing the formality or scope of certain others. Other methods may also be appropriate if agreed upon by both the Government and the contractor prior to program start. Organizations considering fast track or abbreviated programs should have demonstrated successful experience with similar technologies

and have a mature, defined development process to minimize risk. Fast track programs must also have a clearly defined, stable set of requirements. *[In other words, the objective does not have to solve world hunger, but rather to feed a few starving beggars.]* Abbreviated life cycle strategies may be most appropriate when a program can proceed as an **Engineering Change Proposal (ECP)** to an existing contract, or where substantial familiarity exists and/or minimum risk is evident.

Concurrency is a fast track method where development and operational testing are combined with a concurrent follow-on development and initial production phase. With this method, government involvement are often limited. Lessons-learned from the General Accounting Office (GAO) about employing fast track methods have included the following recommendations:

- Make an initial detailed assessment of the technical risks involved in individual subsystems, as well as, in the integration of those subsystems into a workable system, with explicit focus on whether the technology being attempted is compatible with an accelerated acquisition strategy.
- Build into the strategy provisions for adjusting schedules and other program facets if technical difficulties occur.
- Assess the system's technological progress periodically to see if it is still compatible with the planned acceleration. If technical progress is no longer keeping pace with the acceleration, the strategy must be adjusted to bring it in line with the technology.
- Ensure that the strategy provides for testing and evaluation sufficient for decision-makers to identify any major shortcomings in the system's operational suitability and effectiveness which must be resolved before initial production is approved. [GAO86]

NOTE: See Chapter 7, *Software Development Maturity*, for an in-depth discussion on the importance of the contractor's software development process, methodologies, tools, and capabilities to the success of your program. The software measurement life cycle is described in Chapter 8, *Measurement and Metrics*. Words to include in your Request for Proposal (RFP) requiring that offerors provide you with adequate information for proposal evaluation are found in Chapter 13, *Contracting for Success*. How to evaluate offerors' proposals so you select the contractor

with the "*best process*" as well as product is also found in Chapter 13. A description of the Cleanroom engineering life cycle is found in Chapter 15, *Managing Process Improvement.*

REFERENCES

[BOEHM88] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May 1988

[GAO86] General Accounting Office, "Sergeant York: Concerns About the Army's Accelerated Acquisition Strategy," Report to the Chairman, Committee on Government Affairs, United States, Senate, GAO/NSIAD-86-89, May 1986

[GSA91] Alternatives to Grand Design for Systems Modernization, Information Resources Management Services, US General Services Administration, Washington DC, April 1991

[HUMPHREY95] Humphrey, Watts S., personal communication to Capt Joseph Stanko, September 15, 1995

[MARSHALL51] Marshall, BGEN S.L.A., The Armed Forces Officer, U.S. Printing Office, Washington, DC, 1951

[MONTGOMERY58] Montgomery of Alamein, Field Marshall Viscount, Bernard Law editor, The Memoirs of Field Marshall Montgomery, The World Publishing Company, Cleveland, Ohio, 1958

[PASSMORE94] Passmore, John M., "Life Cycle Methodology Offers Software Starting Point," *Signal*, March 1994

[PATTON47] Patton, GEN George S., War As I Knew It, Houghton Mifflin Company, Boston, 1947

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[PRESSMAN93] Pressman, Roger S., "Understanding Software Engineering Practices: Required at SEI Level 2 Process Maturity," Software Engineering Training Series, Software Engineering Process Group, July 30, 1993

[QUANN95] Quann, Eileen., personal communication to Lloyd K. Mosemann, II, September, 1995

[ROYCE70] Royce, Winston W., "Managing the Development of Large Software Systems", *IEEE*, WESCON, 1970

[WOODWARD89] Woodward, Herbert P., "Ada: A Better Mousetrap," *Defense Science*, November, 1989

PART II
Engineering

Blank page.

CHAPTER

4

Engineering Software-Intensive Systems

CHAPTER OVERVIEW

The process you employ to develop software is key to program success. The accepted, proven process embraced by DoD is ***software engineering****. In this chapter you will learn that software engineering presents a practical solution towards improving software quality, controlling software cost, and ensuring that software technology keeps pace with hardware. One of the first steps in the engineering process is domain engineering which facilitates a product-line development strategy, opens up opportunities for reuse, and is integral to the requirements analysis and specification process. A form of domain engineering is* ***information engineering*** *where the goal is to maintain a stable database, while the software and procedures that automate the database are designed to accommodate change.*

Three subject areas encompass the discipline of software engineering. These are: its elements (what makes up the discipline), its ***goals*** *(what is to be achieved), and its* ***principles*** *(what rules must be applied to achieve the goals). Software engineering elements are:* ***methods, tools,*** *and* ***procedures****. Software engineering goals are:* ***supportability, reliability, efficiency****, and* ***understandability****. To achieve these goals, a set of principles must be applied to the engineering process. These principles are:* ***abstraction, information hiding, modularity, localization, uniformity, completeness,*** *and* ***confirmability****.*

Never lose sight of the fact that software is always part of a larger system. Whether it is embedded in the microchip of a cruise missile warhead or stored on the hard drive of a PC, it never exists as a separate entity. Therefore, software engineering must be discussed in the same context with systems engineering. ***Systems engineering*** *concentrates on the coordination, compatibility, and integration of the two primary parts of a system, its hardware and software. One approach to systems engineering is* ***integrated product development (IPD)*** *where diverse technical disciplines are brought together as a cohesive unit and teamwork is the bottom line. IPD teams work to minimize weaknesses and build upon each other's strengths to optimize product quality and the development process. One facet of IPD is the* ***concurrent engineering*** *approach now being employed on the F-22 program. Not only does the integrated team include people from government and industry involved in the aircraft's development, it includes people who normally would not know about the weapon system until it is fielded, the pilots (users) and the crew chiefs (maintainers).*

Blank page.

CHAPTER

4

Engineering Software-Intensive Systems

ENGINEERING IS THE KEY

As we mature into the Information Age, the same forces that assured success during the Industrial Revolution are driving how we produce software. The world's industrial giants attained their status through superior mass production processes developed through advanced engineering. *Mass demand* and *global competition* are driving software production into the world of engineering, the same as they did for hardware.

Computer **hardware engineering** is quite mature and grew out of the manufacturing and electronic design processes. Within the hardware engineering discipline, "*hardware design techniques are well-established, manufacturing methods are continually improved, and reliability is a realistic expectation rather than a modest hope.*" Unfortunately, software has not advanced nor matured as quickly as the electronic hardware upon which it runs. In computer-based systems, where the hardware component is exceptionally stable with predictable fast-paced advances — the software is usually "*the system element that is most difficult to plan, least likely to succeed (on time and within cost), and most dangerous to manage.*" [PRESSMAN92]

New trends in development, however, are gradually removing the riskiest component stigma from software. By applying the engineering discipline that matured hardware beyond the risk threshold, software can now achieve expected levels of reliability, maintainability, and reusability. **Software engineering** is maturing software development, which has been historically characterized as a cottage industry populated by artisans, craftsmen, and skilled maverick

developers. **Engineering discipline** is transf production into a mighty industrial machine ch finely-tuned engineering process that predictably mass produces reliable software, on time, at compe quickest, cheapest, highest quality way to build s make mistakes during its development, and not to than once. Through years of experience and a wel process, world-class software developers have lear right things, the right way — the first time, eve **software engineering discipline** is essential for [ZELLS92]

> *Above all, discipline; eternally and inevitabl Discipline is the screw, the nail, the cement, nut, the bolt, the rivet that holds everything tigh is the wire, the connecting rod, the chain that Discipline is the oil that makes machines ru that makes parts slide smooth, as well as makes the metal bright. The principle of dis is divinely simple; you lay it on thick and fast,*
> — Private Gerald Kersh [KERSH90]

Software engineering discipline cannot be ignored; thick and fast — all the time. It must be institution life cycle. For large military software systems, tl difficult because most requirements are based on su and tactical demands, are dynamic, evolve ov troublesome to precisely define. Developers and verifiers) must have procedures to identify and rem requirements definition and design before they a code. Quality can only be accomplished thro application of software engineering discipline and p to ensure that ***quality is the norm — not the excep*** learn throughout these Guidelines, successful prog strives for both process and product quality throug

- Assess process and product status,
- Foster early process and product error identificati and
- Continuously improve processes and product prevent defects.

CHAPTER 4 Engineering Software-Intensive Systems

The implementation of a disciplined engineering process for software is a complex process. Software engineering for a system interacts with, and is dependent upon, related domain engineering, information engineering, hardware engineering, and systems engineering activities that occur in the production of a total, integrated system. Figure 4-1 illustrates (on a high level) the relationships among these engineering disciplines.

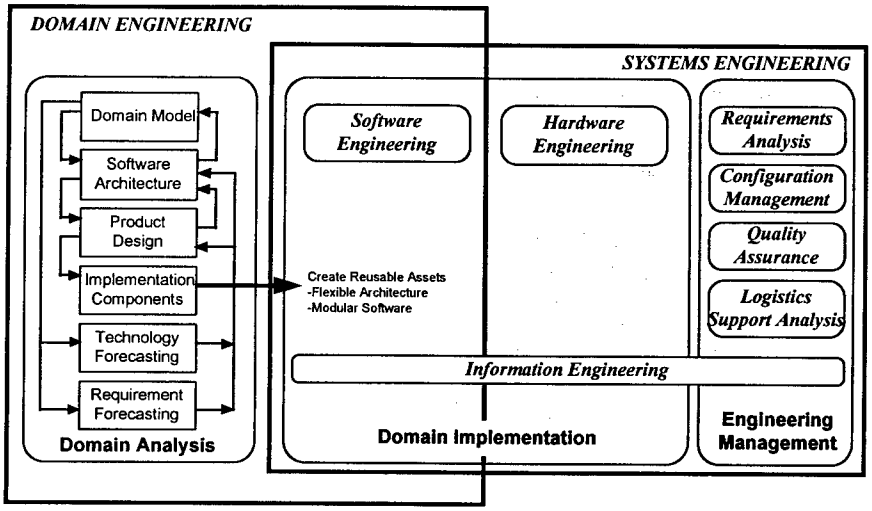


Figure 4-1 Total Quality Engineering

WHAT IS DOMAIN ENGINEERING?

Domain engineering refers to the techniques (i.e., methods and processes) used to engineer a family of similar or related systems (i.e., a domain or product-line). The focus of domain engineering is to capture engineering knowledge (requirements, architectures, components and other life cycle artifacts) within a particular domain for use on future or concurrent programs. This knowledge (captured by models, architecture specifications, etc.) is then used to configure a system architecture and develop (or select) reusable components based upon previous requirements analyses, design, coding, integration and testing efforts. [CARDS94]

Domains are groups of related systems sharing a set of common capabilities. Domains can be described pictorially as having either vertical or horizontal relationships among each other, as illustrated in

Figure 4-2. A **vertical domain** is a specialized class of system, such as an information system, command and control, or embedded weapon system. **Horizontal domains** consist of general software functions applicable across multiple vertical domains. These can include user interfaces, common algorithms (e.g., data structures, strings, matrices, lists, stacks, queues, trees, graphs), common mathematical solutions (e.g., linear systems applications, integration, differential equations), and software tools or graphics packages. Although the domain engineering steps are presented here as sequential activities, in practice they are highly iterative. Major domain engineering steps include:

- Domain identification and scoping,
- Domain analysis,
- Domain design, and
- Domain implementation.

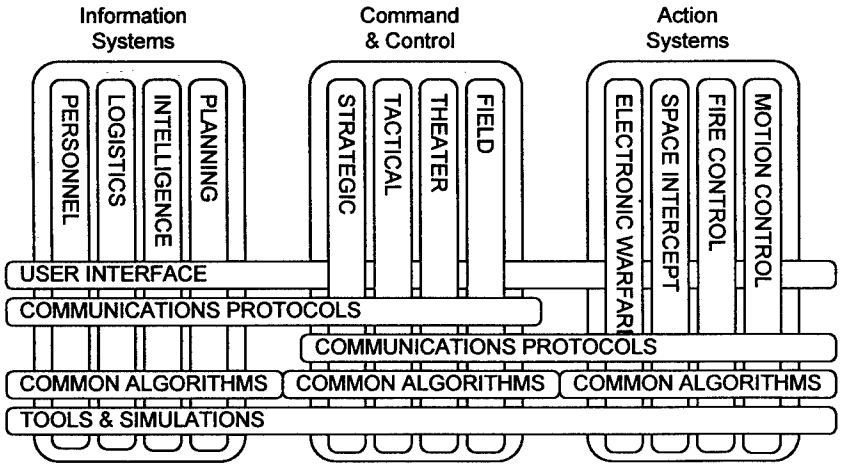


Figure 4-2 Vertical and Horizontal Domains

Domain identification is an iterative process to capture the overall context and definition of the domain. In effect, it defines a domain boundary and models the interfaces among the subject domain and external ones. During **domain analysis**, domain context and definition are refined into detailed domain models showing systems requirements commonality and variability. During **domain design** and **domain implementation**, definition refinement continues to produce generic architectures and reusable components which satisfy domain requirements. Once the domain engineering activity is "*complete*,"

domain information and products must evolve in response to changing requirements and technology.

Domain Identification

The **domain identification** step is important to overall success. Understanding the subject domain and customer needs derived during this phase, drive the entire domain engineering effort. Domain identification defines domain boundaries, interfaces, and dependencies. The knowledge gained during domain identification provides domain analysts with a common understanding of:

- Domain scope (inclusion or exclusion of domain applications),
- The relationship of the subject domain with other domains,
- The relationships among domain applications, and
- The inputs/outputs to and from the domain.

During domain identification, a number of domain-representative systems are identified. These "*exemplar*" systems highlight common, variable domain requirements. The number of exemplars must reflect the level of effort required to conduct the modeling effort with schedule constraints. Note that domain analysis addresses more than the exemplars—it also considers other information such as technology trends and anticipated future requirements.

Domain Analysis

Domain analysis captures and models requirements information across a particular domain. Domain analysis is the process of identifying, documenting, and modeling common, variable requirements among domain systems. Domain analysis techniques include interviews, documentation review, and reverse engineering, to identify and categorize (i.e., model) domain requirements. Other inputs, such as enterprise models (e.g., data models in IDEF1X and operational models in IDEF0), are used during domain analysis. The resulting domain model(s) form a domain problem space (or domain requirements) representation. These models provide a domain perspective of domain systems in terms of data (or objects), functional capabilities, and control or (behavioral aspects). Along with these perspectives, a standard domain vocabulary is developed. The products of domain analysis vary depending on the analysis method used and typically include the following:

- **Information model.** This provides the domain data (object) perspective. During this activity, domain data requirements, essential for implementing domain applications are represented. Variability among exemplars is represented in the information model through alternative objects and/or attributes. Information entities are traced back to the exemplar sources from which they are derived.
- **Feature or functional model.** This captures the end-user's understanding of domain application capabilities through a functional domain systems perspective. Alternative feature commonality and variability among the different exemplar systems are represented in the domain model. Features are categorized and traced back to their exemplar sources.
- **Operational model.** This identifies domain application control and data flow commonalities and differences from a behavioral perspective. This activity abstracts and then structures common domain functions, features, and sequencing into an abstract operational model from which individual application control and data flow are derived.
- **Domain dictionary.** A useful product of domain analysis, this defines the terms and/or abbreviations used in describing domain features, their textual description, and domain entities. The dictionary helps alleviate miscommunication by providing a central location for domain information users to search for unfamiliar terms and abbreviations or for definitions of terms used differently or specific to the domain.

Domain Design

The **domain-specific software architecture** (DSSA) is the foundation of systematic reuse *[discussed in Chapter 9, Reuse]* and the maturing of software engineering maturity. The DSSA provides the high-level design for all domain (or product-line) systems and establishes the context for high-leverage, large-scale reuse. The domain model, created in the domain analysis step, is used during **domain design** to derive the DSSA, which specifies a set of solutions to the requirements represented in the model. The DSSA accommodates domain model requirements variability by capturing context drivers leading to alternative solutions. [KOGUT94] The DSSA identifies:

- **Component classes** are derived through partitioning overall system functionality, as captured in the domain model. A component class represents a category of components with similar functionality

(e.g., DBMS or Geographic Information Systems). Each functional requirement captured in the domain model is allocated to one or more component classes. Component class variability reflects the variability captured in the domain model by specifying alternative and optional classes.

- **Connections** describe how component classes are linked. Typical connections specifications include data flow, direction, and type (e.g., SQL query, protocol). Alternative connections result from variability in domain requirements.
- **Constraints** describe component class characteristics allocated from the domain model and implied by the architecture. That is, the constraints highlight functionality derived from the domain model, as well as the functionality dictated by component class connections. Because of the variability captured in the domain model, it is necessary to specify alternative and optional component class constraints.
- **Rationale** facilitates selection among reusable components. For example, suppose a more expensive DBMS will provide a faster response. This provides the rationale for choosing the more expensive DBMS when response time is critical.

Key to domain design is maintaining traceability between the derived architectural solution and domain modeled requirements. Domain designers can use this traceably to develop an initial systems architecture and to select or build a set of reusable components that best fit the new system's requirements. This allows selection of specific architectural solutions based on user/developer selection of specific domain requirements. This forms the basis for qualified components composition or new component specification and development based on constraints specified in the architecture.

Domain Implementation

Domain implementation refers to: (1) the process of creating new components or modifying existing components for a DSSA component class; and (2) altering components in response to changes in requirements or the detection of defects. These domain assets can be employed or modified to suit new systems development within the domain. Domain implementation can also include the development of automated tools that aid life cycle efforts, such as composition tools, generators, and analyzers. [MAYMIR95]

There are four main strategies for domain implementation: generation; new development; re-engineering; and identification of commercial-off-the-shelf (COTS) and government-off-the-shelf (GOTS) software. In addition, for re-engineered software, COTS, and GOTS, there is a separate domain-specific step for software qualification. Some combination of these strategies is employed to complete domain implementation. The cost and applicability of these strategies depends on tool support (especially for generation), the level of domain maturity (for re-engineering), and the availability of COTS or GOTS software. These approaches must be analyzed to determine an appropriate domain implementation strategy. Specific strategies are then developed to fit the needs of the customer. A brief discussion of these strategies follows.

- **Generation**. If a component class is mature and well-defined, generative techniques may be applied to automatically produce reusable assets that fit the generic architecture. Commercial generators are available to support this strategy in several narrowly focused areas. An example is the graphical user interface (GUI) builder. For component classes where commercial tools are not available, it may be necessary to build special-purpose *application generators* tailored to the class.
- **New development**. Reusable assets can be developed, as part of a normal software engineering effort, to satisfy generic architecture requirements. This strategy, frequently employed by domain engineering teams, is suitable when legacy software is non-existent or not suitable for reuse.
- **Re-engineering**. Legacy systems (e.g., exemplars used during domain analysis) may include components *close enough* to the desired functionality and structure to warrant redesign and/or re-implementation to be reusable within the domain.
- **Qualification**. The widespread use of COTS and GOTS results in lower development and maintenance costs for any single user. (In mature domains substantial legacy software may also exist.) Hence, many domain engineering teams are using available software to satisfy DSSA. Within this context, pre-existing software must be evaluated against DSSA requirements. Component qualification assesses how well a particular component fits into a DSSA component class and ensures components are reusable within a given architectural context. Qualification criteria for evaluating reusable components are dependent on domain characteristics and user needs.

If new software is chosen as the domain implementation strategy, DSSA requirements are used in creating new reusable software assets. Several efforts have focused on developing generic guidelines for creating reusable software (hence, the name "*design-for-reuse*"). However, guidelines differ based on development methodology (object-oriented, functional decomposition, etc.). An appropriate software development methodology must be used to create reusable software assets. Domain engineering provides necessary background for tailoring *design-for-reuse* guidelines for a selected development methodology.

Benefits of Domain Engineering

In September 1991, the Air Force and the Advanced Research Projects Agency (ARPA) selected the Air Force **Space Command's Space Command and Control Architectural Infrastructure (SCAI)** program as the Air Force Demonstration Project for **Software Technology for Adaptable and Reliable Systems (STARS)** megaprogramming concepts. Demonstration programs were also awarded to the Army and Navy.

NOTE: See Addendum A to this chapter, and Chapter 9, *Reuse*, for more information on the SCAI effort.

The demonstration program's goals were to show the feasibility of using an architecture-based, product-line approach to system development. The SCAI program realized benefits in the areas of productivity, error reduction, and cost savings. Productivity went from 175 lines-of-code (LOC) per month to over 1,700 LOC per month. Defects decreased from 3+ errors per 1,000 LOC to about 0.35 errors per 1,000 LOC. Cost per 1,000 LOC also decreased from the typical $140+ to about $57. These benefits were realized because of the domain engineering approach.

For additional information on domain engineering, or assistance in selecting and implementing appropriate domain engineering methods, please contact one of the following organizations *[see Volume 2, Appendix A for addresses and phone numbers, and Appendix B for Web addresses]*:

- AF/Comprehensive Approach to Reusable Defense Software (CARDS) Program,

- DoD/Software Reuse Initiative (SRI), and
- Software Engineering Institute (SEI).

NOTE: See Chapter 9, *Reuse*, for detailed discussions on domain-specific reuse programs and repositories.

WHAT IS SYSTEMS ENGINEERING?

The first formalization of **systems engineering** for military development occurred in the mid-1950s on ballistic missile programs. [DSMC90] Since then, the systems engineering discipline has evolved to encompass both technical and management processes, and has expanded its applicability to cover the entire life cycle of a software-intensive system. A technically-oriented definition of systems engineering is:

> *an interdisciplinary approach encompassing the entire technical effort to evolve and verify an integrated and life cycle balanced set of systems people, product, and process solutions that satisfy customer needs.* [EIA632]

Army Field Manual 770-78, *Systems Engineering* (1979), provides a definition, not specific to any particular industry segment, that emphasizes the ***leadership role*** systems engineering plays in integrating other disciplines. It defines systems engineering as:

> *The selective application of scientific and engineering efforts to:*
>
> - *Transform an operational need into a description of the system configuration which best satisfies the operational need according to the measures of effectiveness;*
> - *Integrate related technical parameters and ensure compatibility of all physical, functional, and technical program interfaces in a manner which optimizes the total system definition and design; and*
> - *Integrate the efforts of all engineering disciplines and specialties into the total engineering effort.*

Whichever definition you prefer, both have the same goal — to effectively balance system elements by integrating them into a complete system that meets customer needs. Systems engineering is not a one time or single phase effort. It is an essential activity throughout the system's life. During the early planning phase it assures flexibility and supportability are built into the design. In later years, it aids in smooth, effective change implementation and modification, often adding value and prolonging the system's life, as illustrated on Figure 4-3. [EIA632]

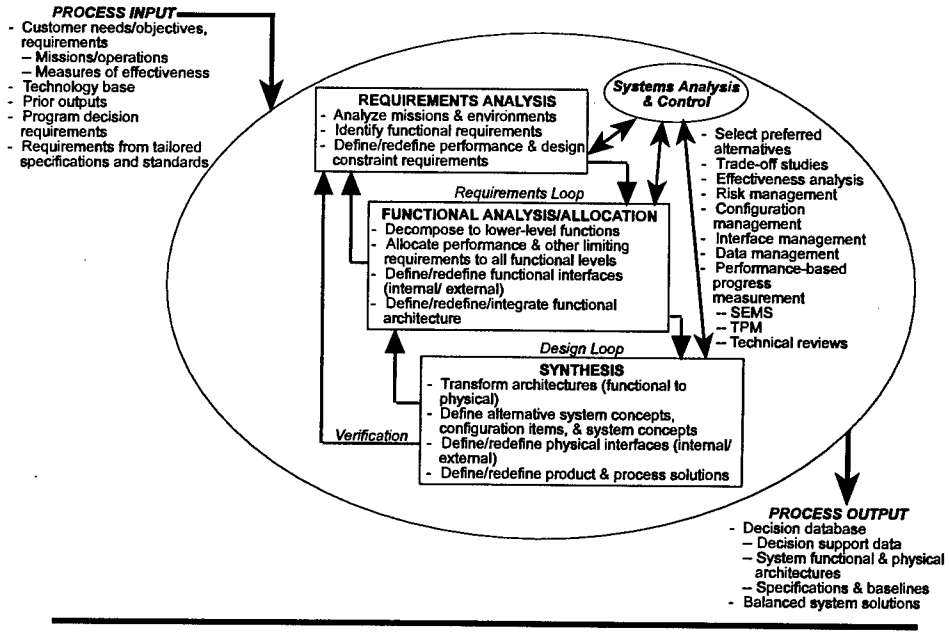


Figure 4-3 Systems Engineering Process [EIA632]

Applied iteratively throughout the system life cycle, the systems engineering process has four elements the **Electronics Industry Association (EIA) Standard 632** describes as:

- Translating stated operational requirements into an integrated product design using a systematic, concurrent approach;
- Transitioning multi-disciplinary technical inputs (including concurrent engineering of manufacturing, logistics, and testing) into a coordinated effort to meet program cost, schedule, and performance objectives;

- Ensuring functional and physical interface compatibility so system definition and design meet all hardware, software, facilities, people, and data requirements; and
- Establishing a risk management program to reduce risk early through system element tests and demonstrations.

The exact activities of the systems engineering process should be documented in a **Systems Engineering Management Plan (SEMP)**, while progress towards completion of these activities should be identified and tracked using **technical performance measurements (TPMs)**. *[Further information on these can be found in EIA 632.]*

NOTE: You are urged to obtain a copy of EIA Standard 632 or IEEE 1220, which were developed from MIL-STD-499B, and follow the guidance found therein. See Volume 2, Appendix A and Appendix B for information on how to obtain these standards.

Of a system's components (i.e., people, products, and processes), two of the more notable products are hardware and software. Systems engineering takes both into account, giving each equal weight in analysis, tradeoffs, and engineering methodology. In the past, the software portion was viewed as a subsidiary, follow-on activity. The new focus in systems engineering is to treat both software and hardware concurrently in an integrated manner. At the point in system design where the hardware and software components are addressed separately, modern engineering concepts and practices are employed for software, the same as they are for hardware. [MOSEMANN92[1]] *[Refer to Standard Systems Group, Systems Engineering Process (SEP), Handbook 700-10. See Volume 2, Appendix A for information on how to obtain a copy.]*

Figure 4-4 illustrates how systems engineering, hardware engineering, and software engineering are concurrent processes. ***The primary role of systems engineering is to ensure that the many diverse elements comprising a system are compatible and ready when needed.*** This avoids the situation in which the hardware or software, when integrated into the system, fails to function harmoniously with other system components. Systems engineering concentrates on comprehensive planning and coordination throughout the development process to ensure integration problems are minimized and that final system implementation fulfills all mission requirements. Different

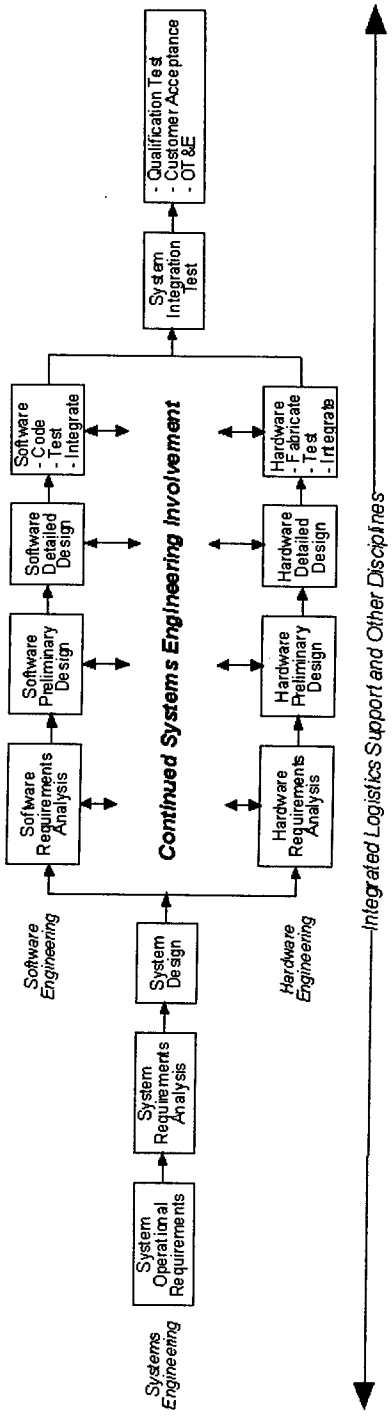


Figure 4-4 Relationship between Systems, Hardware, and Software Engineering

approaches have evolved to implement the systems engineering process. One approach used by DoD is **integrated product development (IPD)** that focuses on the abatement of integration issues.

Integrated Product Development (IPD)

Integrated product development is *"a team approach to systematically integrate and concurrently apply all necessary disciplines throughout the system life cycle to produce an effective and efficient product or process that satisfies customer needs."* [WAGNER95] The key ingredient in IPD is ***teamwork***. IPD provides a technical-management framework for a multi-disciplinary team (comprised of multiple specialties) to define the product. The team includes users (both operational and support) to better address their needs and ensure developers consider all aspects of the system life cycle. IPD emphasizes upfront requirements definition, tradeoff studies, and the establishment of a change control process for use throughout the entire life cycle. This life cycle emphasis is why, according to Captains Gary Warner (USAF) and Randall White (USAF), the F-22 program refers to their IPD teams as *integrated product teams* (IPTs). The term "*development*" is omitted because the IPT continues into the operation and support phase by handling modifications and systems upgrades. [WARNER95] An example of a multi-disciplined IPD team is illustrated on Figure 4-5.

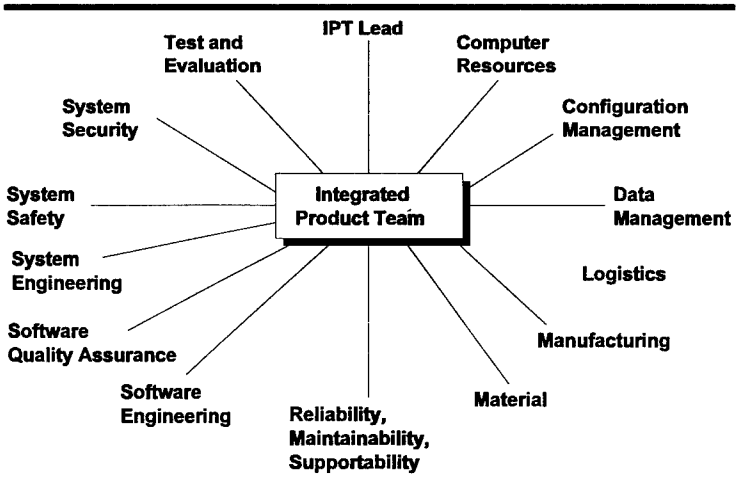


Figure 4-5 Example Integrated Product Team Members

Figure 4-6 illustrates the concept of IPD during the systems design phase. Of course, Figure 4-6 is mainly conceptual, as several iterations through each filter step are often required. Four integration filters are shown in the overall process. As information is taken into the *traditional discipline filter*, emphasis is placed on traditional design techniques (such as structural stress analysis) required at any given design stage. Traditional design engineers rely heavily on current technology. At the same time, design documentation is developed and/or modified by *engineering specialists* who establish requirements independent of the emerging traditional design. They also review and modify the traditional design output. All requirements are then filtered by the unique demands of system products. Subsequently, requirements are described by specifications and drawings (or in some cases, prototypes) filtered through the user group to determine whether they satisfy needs.

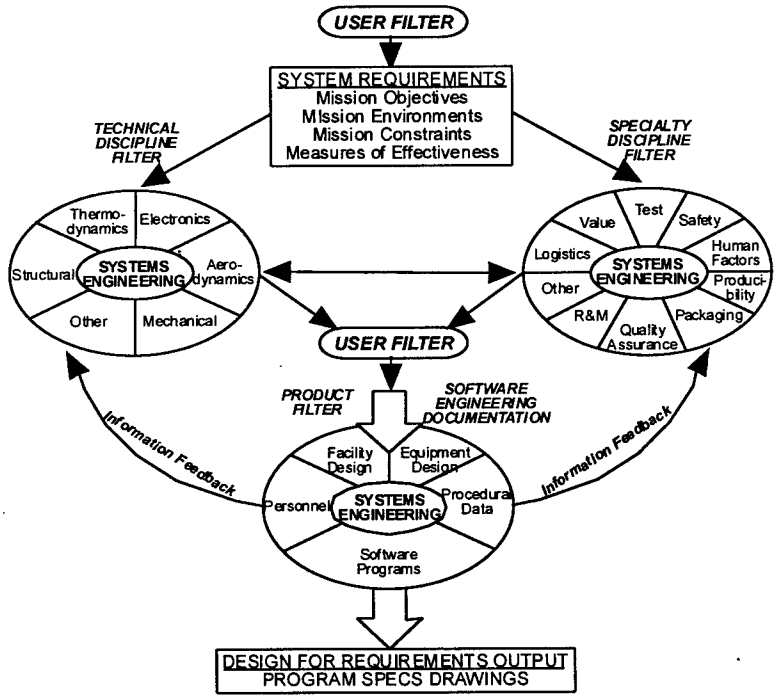


Figure 4-6 Integrated Product Development Process

Concurrent Engineering

Concurrent engineering *[not to be confused with concurrent acquisition described in Chapter 1, Software Acquisition Overview]* is one of several systems engineering disciplines within the IPD approach used to define requirements and manage system acquisition and development. Concurrent engineering is *"a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support."* [WAGNER95] Concurrent engineering, in this context, is the coordination, integration, and sequencing of the multi-discipline engineering activities that must occur to produce a major software-intensive system. The following summarizes concurrent engineering benefits achieved in three industrial applications:

> *The Boeing Commercial Airplane Group is using it [concurrent engineering] to develop the giant 777 transport and expects to release design drawings a year and a half earlier than with the 767. John Deere & Co. used it to cut 30% off the cost of developing new construction equipment and 60% off development time. AT&T Co. adopted it and halved the time needed to make ... an electronic switching system.* [SHINA91]

Concurrent engineering ensures that people from many disciplines collaborate throughout the life of a product (*from-cradle-to-grave*) to ensure it performs to user's needs and requirements. For example, people from engineering, software, operations, maintenance, and manufacturing work as a team from program onset to anticipate problems and bottlenecks and remove them early. This approach avoids delays in fielding the product and prevents costly operational failures. Participation by contracts and logistics personnel also ensures a smooth acquisition process, low product cost, and availability of reliable supplies of parts and materials.

Automated design tools, computer-aided manufacturing systems, and information management tools are commercially available for concurrent engineering applications. A centralized graphical representation can help the team visualize key elements and relationships to achieve a multi-disciplinary design solution. Automated products enable the storing of supporting data so the results of historical design efforts can be applied to the task at hand.

This hastens product improvement efforts beyond many life cycle iterations. [SHINA91]

An example of concurrent engineering in practice is the F-22 program. **Colonel Robert Lyons, Jr.**, former co-leader of the F-22 System Program Office Avionics Group, explained that IPD is being employed throughout the F-22 program office, where he says they are using an expanded version of IPD with concurrent engineering. The program office is uniting program managers, as well as specialists in contracts, cost analysis, test, safety, logistics, and quality assurance to oversee product development. Lyons says that, "*Already in this program we [Government and industry] have laid on the table information that in other programs people wouldn't have heard about for several years.*" He explains that the beauty of including everyone affected by the development in areas other than their own is that all program concerns and requirements are identified and addressed upfront. [LYONS92] Everyone, in this case, also includes the F-22's customer, Air Combat Command (ACC). According to Captains Wagner and White, ACC is active in the F-22 Weapon System concurrent engineering effort, having local representatives who are *"active team members and provide on-the-spot inputs for requirement issues."* This inclusion of the customer into the concurrent engineering team *"kept the user in the loop and provided a quick way of obtaining guidance on requirements."* [WAGNER95] A recent assessment of IPD on the F-22 identified several key factors needed for successful implementation:

- Implement IPD from the beginning of the program with extensive planning to make it happen;
- Train and educate the team members on IPD, including new personnel coming in mid-program;
- Enhance communications with co-location of team members and use of electronic mail;
- Structure the system program office (SPO) to reflect the IPD system breakdown;
- Have the necessary integrated management tools to do the job [these include technical performance measurements, integrated master plans (or systems engineering management plans), and integrated master schedules (or systems engineering management schedules)]; and
- Establish an **analysis and integration (A&I)** team to integrate IPD team efforts to ensure *"I"* stands for *integrated* and not *independent.* [WAGNER95]

The Case for Software Engineering

The forces driving DoD towards *engineering* our software are primarily economic. **Lloyd K. Mosemann, II,** former Deputy Assistant Secretary of the Air Force (Communications, Computers, and Support Systems) explains that, "*military software must be engineered. There is too much of it and systems are too large to develop cost-effectively using the hand-tooled, cost-insensitive 'software-as-art' model.*" [MOSEMANN92[2]] He further states that, "*The definition and institutionalization of software engineering in the Air Force is now our highest priority.*" [MOSEMANN91] From the DoD perspective, **Paul Strassmann**, former Director of Defense Information (ASD/C3I), reinforced the case for software engineering by declaring, "*The No. 1 priority of DoD, as I see it, is to convert its software technology capability from a cottage industry into a modern industrial method of production.*" [STRASSMANN91]

To understand what we mean by ***software engineering***, the **Software Engineering Institute (SEI)** examined the mechanical and civil engineering disciplines which evolved from *ad hoc* solutions to *engineered* ones based on scientific principle. By **scientific principle** we mean:

> *an attempt to explain a certain class of phenomena by deducing them as necessary consequences of other phenomena regarded as more primitive and not in need of explanation.* [McGRAW89]

As practitioners within a discipline accept new explanations, the discipline shows a progression from crafted *ad hoc*, solutions to a formal, codified body of knowledge. Scientific principles, in the form of proven mathematical statements, are developed to explain and predict results. Initial solutions establish the foundations for creating new instances predicated on scientific principles. New and larger problems can then be addressed based on initial solutions. During this evolution, the state-of-the-practice constantly improves. Software engineering involves improving the practice through the codification of collective knowledge and experience. [HOLIBAUGH92]

NOTE: See the *Scientific American* article, "Software's Chronic Crisis," at the beginning of this volume for a discussion on the evolution of a craft to an engineering process.

In contrast to engineered solutions, crafted solutions are unique and problem-specific and the experience base is usually limited to that of the practitioner. **Eileen Quann**, president of Fastrak Training, Inc., equates the differences between software-as-art (or craft) and an engineered product to the different approaches required when building a dog house, a family home, and a skyscraper, as illustrated on Figure 4-7. In each case, building construction generally consists of assembly functions. For all three structures, a floor, walls, a roof, a door, and windows must be built.

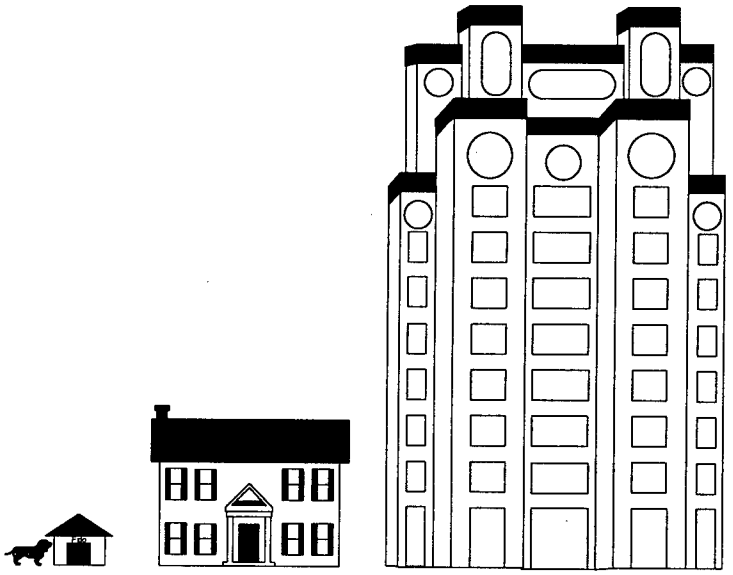


Figure 4-7 Order of Magnitude Between Software Engineering and Software-as-Art

The differences among the construction projects are found in the skills and tools needed to accomplish each job. Your teenage son can build a doghouse with a few nails and some wood. However, he is not qualified to build your family home which requires craftsmen skilled

in reading blueprints, plumbing, wiring, roofing, flooring, insulation, and inspections. Similarly, the same craftsmen are not usually qualified to build a skyscraper, which requires additional skills in such areas as joining steel beams, installing tremendous amounts of glass and concrete that must withstand enormous physical stresses, and electrical wiring with demands much greater than a normal house. More importantly, while your teenage son may be able to both design and construct the doghouse, ***when a program reaches the size of something like a skyscraper, design engineers must have more experience and knowledge than is required of a normal engineer/ craftsman***. To design a large-scale building containing immense walls of glass, steel beams, and concrete, a design engineer must be an educated professional, proficient in topics such as the physics of structural stress. They must also have expertise in additional areas such as elevator dynamics, optimum space utilization, environmental power plants, and emergency and handicapped access requirements. Therefore, ***it is just as important that your engineering design team be appropriately trained and experienced as it is that your construction personnel have the right skill and experience level.***

Software engineering is also required for economic reasons. Consider the fact that ***the cost to support deployed DoD software system comprises 60% to 80% of total life cycle costs***. [Ada/C++91] The high cost of software support stems from products often so unique and hand-crafted no one other than the original developer can understand them. Supporting agencies have had to start from scratch when upgrading or enhancing the software they are responsible to maintain. Defects, not discovered until the software is deployed, are at times impossible to correct. Strassmann warns these practices are no longer acceptable, "*Because we don't have the cash anymore to reinvent and reinvent and reinvent exactly the same routine.*" [STRASSMANN92] Figure 4-8 illustrates how software costs have historically been disbursed when software was developed as art. It also shows the difference in spending ratios when software is engineered. What Figure 4-8 does not show, however, is that the cost pie shrinks when software is engineered because software support costs are substantially reduced.

When you use the structured discipline imposed by the engineering process, costly **software support problems** are addressed upfront. Reliability, maintainability, and supportability are designed ***into*** the system instead of being included after development and deployment. Resources are planned and managed within a total life cycle framework.

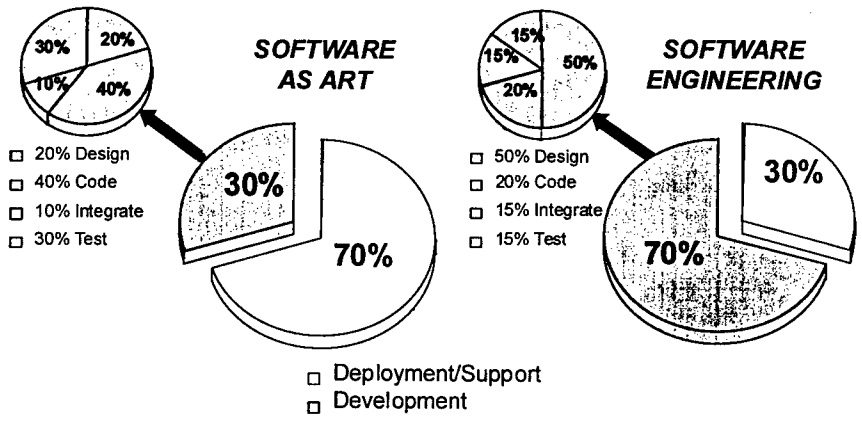


Figure 4-8 Software Life Cycle Costs

Large investments are placed in upfront design and engineering to gain savings over the life of the system.

To state that engineering is a solution to our software problems is no revelation. Simply put, systems and software engineering provide sound, proven discipline for achieving program success. As An Wang, founder of Wang Laboratories, aptly stated, "*Success is more a function of consistent common sense than it is of genius.*" [WANG86] By converting software production from a cottage industry into a modern industrial process, the same benefits can be attained as those gained through the engineering and mass production of hardware:

- Lower unit price,
- Lower maintenance costs,
- Reusable and interchangeable parts, and
- Greater reliability.

Domain Engineering and the Software Engineering Process

Mature engineering disciplines support clear separation of routine problem solving from the research and development required to address unprecedented aspects of systems within a well-defined product-line. Fundamental to such a discipline is the leveraging of a publicly-held, experience-based, and formally transitioned technology base that includes product models (e.g., designs, specifications,

performance ranges) and practice models (tools and techniques to apply the product models). A critical characteristic of mature engineering is that the products built from these models are well-understood and predictable before they are produced. Software engineering state-of-the-practice has yet to reach this level of maturity. Instead of basing new development on a technology base of well-understood models, current software engineering practice tends to start each new application development from scratch with the specification of requirements, and moves directly into design and implementation. By contrast, disciplined software engineering relies on a stable technology base of reusable assets, including requirements, designs, architecture, and software.

Figure 4-9 illustrates the role of domain engineering in establishing a mature, disciplined software engineering process and a product-line development strategy. The stable technology base, specific to the product-line, called the product-line asset base, is created and maintained by domain engineering, which while distinct and separate from the application engineering activity, defines, drives, and constrains application engineering. Domain engineering analyzes, selects, and produces the assets to populate the product-line asset base, which captures the commonality and variability across an entire product-line and includes models that facilitate understanding and specialization to a particular system. The application engineering

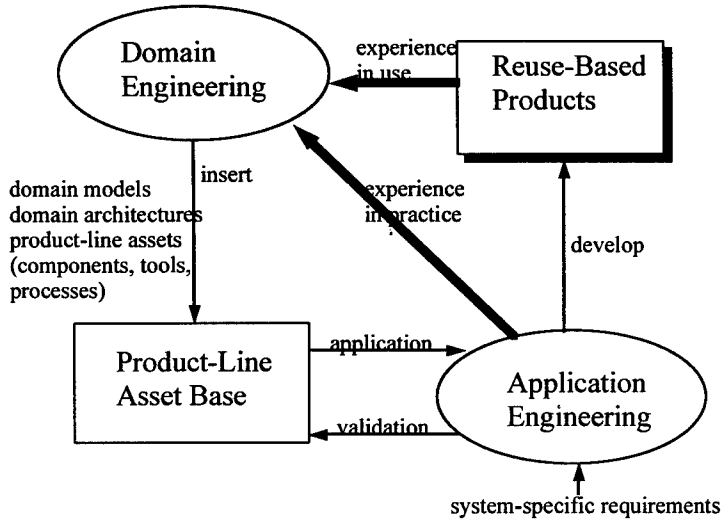


Figure 4-9 Domain Engineering and Software Engineering Discipline

process then uses these products and processes to develop software systems within the product-line. The application engineer draws upon these assets to develop reuse-based products (i.e., software systems). By using well-understood requirements, architecture models, well-documented processes, and high-quality reusable software, the engineer is able to quickly and cost-effectively build more reliable and predictable software systems for the product-line. *[See Chapter 9, Reuse, for further discussion of the product-line approach.]*

The separation of domain engineering from application engineering highlights the need and significance of developing reusable *corporate assets* including domain models, architectures, processes, and components. The application engineering function then focuses on using, validating, and extending this technology base, instead of beginning with a blank sheet. In addition to creating the initial set of domain assets, domain engineering processes continues to add and enhance the technology base according to the requirements associated with application engineering.

Relationship Among Enterprise Engineering, Domain Engineering, and Application Engineering

There has been a service-wide effort to develop organizational enterprise models, which combined comprise the **DoD Enterprise Model**. While the synergy between domain and application engineering has become better understood, the connection to enterprise engineering has remained weak. Domain engineering represents an intermediate level of abstraction between knowledge captured at the enterprise level and the array of systems developed at the application engineering level. Domain engineering reduces the complexity (and hence the risks) of leveraging enterprise-wide common data and functions in the development of individual applications using classical *divide-and-conquer* techniques. Figure 4-10 (below) illustrates the basic methods (at a very high level) associated with the three major software engineering processes: enterprise engineering, domain engineering, and application engineering. Each of these processes attacks the problem space, the solution space, and the implementation space at different levels of abstraction. Application engineering is concerned with a single system/application, whereas domain engineering takes into account multiple similar or related systems; and enterprise engineering looks at an entire enterprise's (organization's) high-level data and operational needs.

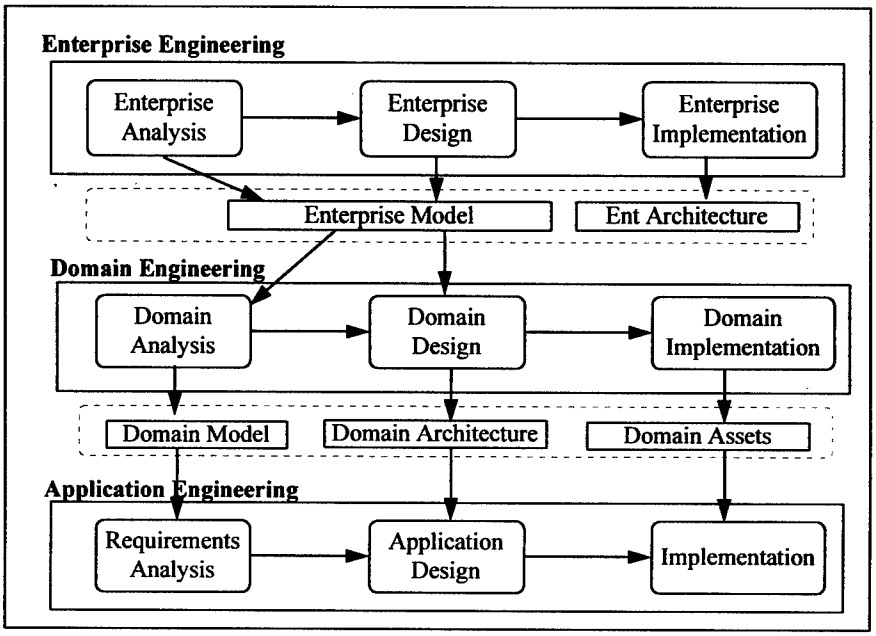


Figure 4-10 Three-tiered View of Organizational Engineering Processes

Viewed top-down, the enterprise for an individual organization can be broken down into multiple domains, which in turn can be broken down into multiple applications. Managing and engineering software from these three different levels helps mitigate risks. It also ensures that information and insight developed at higher-levels of abstraction are leveraged at lower levels.

WHAT IS SOFTWARE ENGINEERING?

Mosemann defines **software engineering** in a meaningful context for software managers when he explains,

> *By software engineering, I mean simply the application to software of the traditional engineering process, which encompasses the following kinds of activities:*
>
> - *Iteration between formal analysis and design,*
> - *Heavy use of earlier designs,*
> - *Tradeoffs between alternatives,*
> - *Handbooks and manuals,*
> - *A pragmatic approach to cost-effectiveness, and*

- *Attention to economic concerns.* [MOSEMANN92[3]]

DoD 5000.2-R, *Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs*, applies the traditional engineering process to software. It states that,

> *"Software shall be managed and engineered using best processes and practices that are known to reduce cost, schedule, and technical risks. It is DoD policy to design and develop software systems based on systems engineering principles..."*

The principles include:

- Use of open system concepts;
- Exploiting software reuse opportunities;
- Use of Ada in government-supported applications;
- Use of DoD standard data;
- Selecting contractors with domain experience, a successful past performance, and a demonstrable mature software development capability and process; and
- Use of software metrics.

Additionally, software engineering structures the complexity of software development with a defined set of techniques and methods to measure and control the process. [ZRAKET92] Pressman identifies methods, tools, and procedures as the basic elements necessary to ensure a quality product:

> *[Software engineering is]...an outgrowth of hardware and systems engineering. It encompasses a set of three key elements—methods, tools, and procedures—that enable the manager to control the process of software development and provide the practitioner with a foundation for building high-quality software in a productive manner.* [PRESSMAN92]

Software engineering methods define the technical *how-to's* for software development. Methods cover a range of tasks that include:

- Program planning and estimation *[discussed in detail in Chapter 12, Planning for Success],*
- System and software requirements analysis, *[discussed in detail in Chapter 14, Managing Software Development],*
- Architecture design, *[discussed in detail in Chapter 14, Managing Software Development],*
- Algorithm procedure and data structure development, *[discussed in detail in Chapter 14, Managing Software Development],* and
- Coding, testing, *[discussed in detail in Chapter 14, Managing Software Development],* and support activities *[discussed in detail in Chapter 11, Software Support].*

Software engineering tools give automated (or semi-automated) support to the methods. A variety of tools support each of the methods listed above. **Computer-aided software engineering (CASE)** is an integration of different tools where information created by one tool can be used by other tools. CASE combines software, hardware, and a software engineering database of information about analysis, design, code, testing, and metrics. *[Software engineering tools recommended for DoD programs are discussed in Chapter 10, Software Tools. Volume 2, Appendix A provides information on how to contact the different tool vendors.]*

Software engineering procedures merge the methods and tools for rational, timely software development. Procedures establish the order in which the methods are applied, deliverables (reports, documents, capabilities, functions) are required, and controls (ensuring quality and coordinating change) are enacted. They also define the milestones needed to evaluate software development progress.

Software engineering discipline consists of defined steps combining the methods, tools, and procedures forming the basis for process improvement activities. These steps are often referred to as software engineering **paradigms** (or models). Paradigms must be selected based on your type of program and application, the methods and tools used, and the constraints and deliverables required. [PRESSMAN92] Figure 4-11 summarizes the components of software engineering.

As with systems engineering, numerous approaches have evolved for implementing the software engineering process. For weapon systems, various approaches are integrated within the systems engineering process. The **IPD** example is one such approach. For MIS,

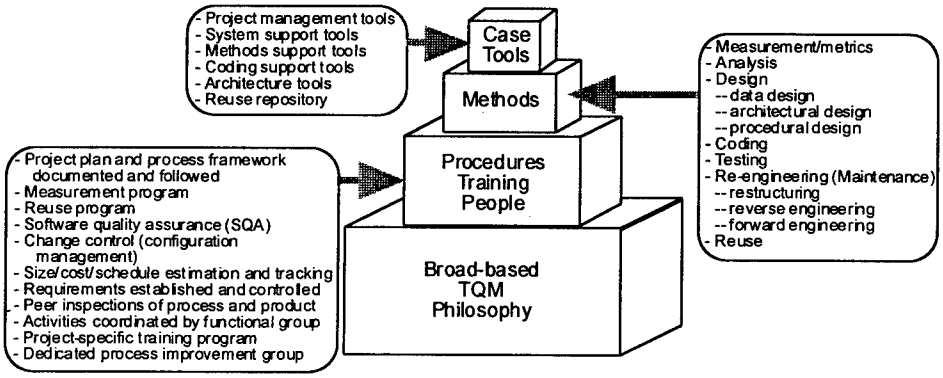


Figure 4-11 Software Engineering Elements

information engineering (IE) and **IDEF** are approaches that implement the software engineering process. What brings the methods, tools, and procedures of software engineering together are its goals and a set of principles that must be accomplished to achieve engineered software. The defined set of **principles,** transcending all engineering efforts, support and implement the **key goals**. They also help with the complexity of managing the development of a major software-intensive system.

NOTE: Because understanding software engineering principles and Ada are interdependent, a discussion on how Ada language features enforce the principles, thus enabling the goals, is found in Chapter 5, *Ada: The Enabling Technology.*

Software Engineering Goals

Remember, the primary goal for all software engineering efforts is for the software solution to meet user needs by fulfilling stated requirements. However, in many large, complex software-intensive systems, requirements often evolve over the life of the system with the greatest costs incurred during the maintenance phase. Given that *change is a constant* in the life cycle, the discipline of software engineering is founded on four main goals, sufficiently inclusive to be unaffected by change throughout the system's life. The main software engineering goals are ***supportability, reliability, efficiency,*** and ***understandability***.

Supportability

Supportability *[discussed in detail in Chapter 11, Software Support]* is the ability to perform maintenance and enhance, upgrade, or otherwise change the software. Component aspects of supportability include maintainability, adaptability, and modifiability. Weapon system software requires a high degree of supportability, as it must be changed to keep pace with evolving threats. Both weapon system and MIS software must also be regularly altered to keep up with the evolution of user, operational, and support requirements. These environmental and evolutionary factors result in *controlled software changes*. Defect correction is viewed as a controlled software change.

As discussed in Chapter 14, *Managing Software Development*, to effectively support a software system, all the *explicit* and *implied* design decisions comprising the solution must be honored. This requires the that design rationale be captured in a manner that software support personnel can use during the system's normal 10 to 30 year operational life of the system. If this information is not captured and considered, new software will end up being patched into the original code by breaking apart the logical basis of the design. Also, if the software is initially poorly designed and constructed, after several block updates the original structure will tend to deteriorate and get lost, complicating future software support efforts. ***Well-engineered software systems are easily supportable and can accommodate changes without increasing the complexity of the original design.***

Reliability

Reliability *[discussed in detail in Chapter 14, Managing Software Development]* is a determinant of system quality and a critical goal where the cost of failure is high (e.g., in terms of equipment replacement or human lives). **Software Reliability** is the probability that the software in a system will perform without failure for a specified period of time in a specified environment. Critical software reliability issues must be addressed early in the design process. Reliability must be built in upfront to prevent errors during conception, design, and development, as well as to recover from failures during operations. ***Reliability can only be designed in!*** It cannot be tested in nor can it be included as a retrofit due to an after thought. ***The goal of optimum system performance is well-engineered software which is 100% reliable.*** If the software can be repaired instantaneously

without disrupting its operation, it also has 100% availability, regardless of how often it fails. [HUMPHREY89]

NOTE: For an in-depth discussion of software reliability, see Volume 2, Appendix O, Chapter 4 Addendum B.

Safety

Field Marshall Sir Archibald Wavell, highly decorated World War II British Army commander and author on the art of war, expressed the military safety issue when he stated:

> *...the soldier's chief cares are, first, his personal comfort;...and secondly, his personal safety; i.e., that he shall be put into a fight with as good a chance for victory as survival.* [WAVELL53]

Software safety *[discussed in detail in Chapter 14, Managing Software Development]* is closely tied to software reliability, and is the guarantee that the system will not fail under stressed operational conditions. Like software reliability, the issues of software safety must be addressed squarely during program planning and development. Ideally, this means the software will be free of defects. However, because software is created by humans, no matter how carefully the system is designed, coded, and tested, the probability for defects caused by human error are always be present. Until we have conquered the human factor through techniques such as highly automated development environments, ***safeguards must be built into all software upon which human safety, and indeed survival, are dependent.*** The cost for building in these safeguards must, therefore, be factored into your cost, schedule, and resource estimates *[discussed in Chapter 8, Measurement and Metrics, and Chapter 12, Planning for Success.]*

While **Defense Nuclear Agency** regulations cover the area of software safety for nuclear weapons, software safety must be a topic of concern especially for all weapon system and C3 systems. One technique for enhancing safety is to employ software **fault-tolerance** methods in critical applications or application segments. One example of software fault-tolerance is the **recovery block** technique, where a failure in the primary program is bypassed by executing an independent alternate program that, hopefully, will execute successfully. Other fault-tolerance techniques are **multi-version**

programming and **exception handling**. *[See Chapter 5, Ada: The Enabling Technology, for a discussion on "Ada's Exception Handling Feature."].* ***It is essential to require stringent fault-tolerance methods such as exception handling for flight control systems, C3 surveillance and sensor systems, and other systems where system aborts requiring restarts are unacceptable and potentially life-threatening.***

Efficiency

Efficiency is an important software capability which refers to the highest and best use of critical resources. Processor cycles and memory locations are considered critical resources. ***Efficiency is a performance requirement that must be addressed during software requirements analysis***. Efficiency can also be achieved during the coding phase — the last point where nanoseconds or bits can be squeezed out of software performance. Three factors should be considered when addressing efficiency requirements: (1) software should be as efficient as required — not as efficient as possible; (2) good design can improve efficiency; and (3) code efficiency and code clarity go *hand-in-hand* and should not be sacrificed for nonessential improvements in performance.

Source code efficiency is a direct result of algorithm efficiency defined during detailed design. As illustrated in the Ada versus assembly example in Chapter 5, *Ada: The Enabling Technology*, many compilers have optimizing features that automatically produce efficient code by breaking down repetitive expressions, using fast arithmetic, and applying efficiency-related algorithms.

In the MIS world, **memory efficiency** is not equated to the minimum memory used. Memory efficiency takes into account the *paging* characteristics of an operating system. Code location or maintenance of functional domains by way of structured components is one way well-engineered software reduces paging, and hence, increases efficiency. In the weapon system software world, memory restrictions are a real and critical concern, although low-cost, high-density memory is rapidly evolving. *[History has shown, ironically, that whatever memory is available is how much the software will need!]* Memory restrictions are generally a product of the size and weight limitations for housing and shielding processors. If system requirements demand minimal memory, compilers must be carefully evaluated for memory compression, or as a last resort, **assembly language** may have to be used. Unlike other software system

characteristics that must be juggled against each other, techniques for **run-time efficiency** can sometimes lead to memory efficiency. The key to well-engineered software with high memory efficiency is ***keep it simple***.

There are two classes of input/output (I/O) efficiency, external and internal. **External I/O efficiency** measures the user interface. Input supplied by the user and output produced for the user are efficient when the information supplied is easily understood. **Internal I/O efficiency** evaluates the I/O directed from one device to another device (e.g., from a computer to a disk or to another computer) or among modules within the same system. This measure is usually expressed in terms of hardware speeds, but the true measure is in ***throughput*** that includes processing time required to process data and transmit it from one module to another. [PRESSMAN92]

Understandability

Understandability is an important goal for the management of complexity. It is the link between the statement of the problem and the corresponding solution. For software to be understandable, it must reflect a natural view of the world. Achieving of this goal involves producing a solution to the stated problem in the form of an effective, **understandable architecture**. Capturing such a structure in software is necessary for it to be supportable, efficient, and reliable.

Different factors make software understandable. Well-engineered software is readable as a result of proper coding and proper documentation (including interface documentation). Well-engineered software also represents an accurate, understandable model of the real world. Understandability is achieved when the data structures (objects) and algorithms (operations) in the software solution are easily distinguished from one another. Understandability is also dependent on the programming language chosen to express the solution. [BOOCH94]

Software Engineering Principles

The goals discussed above are generic in nature and applicable to any software system — large or small. Once you understand these goals, you must employ a structured, disciplined development approach to achieve them. Administering sound engineering practices, based on the following principles, produces solutions that are supportable, reliable, efficient, and understandable.

Abstraction and Information Hiding

Abstraction. As discussed in Chapter 1, *Software Acquisition Overview*, one reason major software-intensive acquisitions fail is our inability to deal with software complexity. **Abstraction** is the software engineering principle for managing ***complexity***. The purpose of abstraction is to separate the essential characteristics of a process, or its data dependencies, from all non-essential details. Abstraction is also performed during software design where the problem is decomposed into increasing levels of detail (or decreasing levels of abstraction). A analogy can be made from the road map example. When you plan a trip across country you start with a map of the United States (high-level). As you go through each state you use a map of the state in which you are traveling (mid-level). As you approach your destination and are looking for a particular address, you use a city map (low-level).

The software engineering process is, itself, an example of the abstraction principle. Each step in the process is a refinement of the abstraction level of the end product — the solution. During systems engineering, software is abstracted to a component of a software-intensive system. During software requirements analysis, the software solution is defined in terms that relate to the problem environment or function it must perform. The level of abstraction is reduced further as you proceed from architectural design to detailed design. Ultimately, the lowest level of abstraction is reached when the source code is written.

As the solution is decomposed into its component parts, each module in the decomposition becomes a part of the abstraction at a given level. Abstraction can be applied to both the algorithms and data in the solution. Thus, the logic of a software solution can be expressed in terminology that describes the problem domain rather than in software-dependent terms. Eventually, the details of expressing the problem will have to be addressed in software terminology — and ultimately in code. However, they can be deferred to lower levels where attention to essential details can be worked and/or reworked without impact on other system levels. Thus, the number of items tackled at one time are reduced to a manageable amount because attention is focused on the current level of decomposition. Abstraction promotes the goals of understandability and maintainability.

Information hiding. Where abstraction separates essential and non-essential details at any given level, the purpose of **information hiding** is to make inaccessible those details that do not affect other parts of the software system. The principle of information hiding is to design modules such that the information contained within a module is inaccessible to other modules having no use for it. *Hiding* means that modularity can be effectively accomplished by defining a set of independent modules. The only information passed among the modules is that which is necessary to achieve functionality.

Abstraction promotes software maintainability and understandability by reducing the number of details a developer is required to know at any given level. It also details the procedural (or informational) entities making up the software. Hiding defines and governs access limitations to procedural information within a given module and any local data structure used by the module. By including information hiding as a design criterion for modular systems, well-engineered software reaps the greatest benefits when modifications are made during testing and later during software maintenance. With most data and procedures hidden from other parts of the application, reliability is enhanced. In addition, inadvertent defects introduced during modifications are less likely to spread to other modules. [BOOCH94]

The benefits of abstraction and information hiding apply to all software engineering goals. Abstraction supports modifiability and understandability by reducing the amount of detail at any given level. Information hiding enhances software reliability, because at each level of abstraction, only essential operations are permitted. Operations that obstruct or confuse the logical structure are also hidden.

Modularity and Localization

Modularity. The principle of **modularity** has been around for almost 40 years and applies to the physical software architecture. Organizing very large applications into discrete, separately named and addressable **modules** allows us to intellectually manage software complexity. Also, with the right selection of module contents, the physical architecture can be made to correspond with the logical architecture, making the overall system more supportable and extendable.

Modules can be *functional* (procedure-oriented) or *declarative* **(object-oriented).** Because reliability must be ***built in***, well-engineered software has well-defined *interfaces* connecting its

modules. No matter how well-defined a module is, it must be able to interact with other modules. **Coupling** is the measure of interface tightness between modules. *Loosely coupled* modules can be treated relatively independently from others, and are easier to interface once integrated. How tightly bound or related the internal module elements are to one another is called **cohesion**. Modules with *strong cohesion* are desirable because their internal components have similar functionality and logical interdependence, making the modules basically self-contained. Self-contained modules are conceptually easier to handle and permit teams of programmers to work independently from each other.

Localization. Applying the principle of **localization** helps create modules with loose coupling and strong cohesion. The principle of localization deals mainly with *physical location*. A module that has strong cohesion has a collection of logically-related resources physically located within it. Localization also implies that modules are as independent of other modules as possible (i.e., well-engineered software has a loosely coupled organization among its modules).

The principles of modularity and localization support the goals of modifiability, reliability, and understandability. In well-structured software, any given module is understandable — independent of other modules. Since design decisions are localized in given modules, modification can be limited to a small set of modules. In addition, if modularization has been successful, there will be limited and looser interconnections among modules. [BOOCH94] This results in greater reliability as defects in loosely coupled modules do not impact the performance of neighboring modules to the extent that tightly coupled ones do.

Uniformity, Completeness, and Confirmability

Abstraction and modularity are the most important principles used to control software complexity. But they alone do not ensure that the software is consistent and accurate. Uniformity, completeness, and confirmability provide these properties.

Uniformity. The principle of **uniformity** directly supports the goal of understandability by ensuring modules use consistent notation and are free from unnecessary differences. Uniformity results from good coding practices with a consistent control structure and calling sequences for operations where logically-related objects are represented the same at any level.

Completeness. The principles of completeness and confirmability support the goals of reliability, efficiency, and modifiability by aiding in the development of solutions that are accurate. Where abstraction extracts the essential details of a given problem set, **completeness** ensures that all important elements are included. Abstraction and completeness guarantee that the modules developed are *necessary* and *sufficient*. Efficiency can be improved because lower-level implementation can be fine-tuned without affecting higher-level modules.

Confirmability. The principle of **confirmability** means the software is decomposable so it can be readily tested, thus enabling modifiable software. The principles of completeness and confirmability are not easily applied. A programming language with strong typing (such as Ada) facilitates the production of confirmable software. Software management tools are also used to ensure software is complete and confirmable.

MANAGING SOFTWARE ENGINEERING

Management is ***the*** key element in the engineering process as it permeates the entire life cycle. To conduct a successful software acquisition program, you must understand the scope of the work to be accomplished, the risks you will incur, the resources required, the tasks to be performed, the milestones to be tracked, the effort (including cost) to be expended, and the schedule to be observed. To be a successful manager, you must understand all facets of your program and rely on educated, experienced software professionals who understand and can implement software engineering process complexities. Sound management starts before the technical work begins, continues as the software matures from a concept to a functional reality, and only ends when you or the system is retired. [PRESSMAN92]

In Chapter 1, *Software Acquisition Overview*, you were told there are three basic activities you must perform as a manager to ensure program success. These activities are:

- You must plan;
- You must manage; and
- You must measure, track, and control.

Having made the software engineering commitment, as policy prescribes, the following items ***must*** be addressed to ensure your program is on the right track and that your developer is ***engineering*** your software. *[See Air Force memoranda, "Software Engineering" and "Air Force Software Policy Objectives," Volume 2, Appendix C.]* These software engineering management activities include and are discussed in detail in Part 2, *Engineering*, in the following chapters:

- Chapter 5, *Ada: The Enabling Technology,*
- Chapter 6, *Risk Management,*
- Chapter 7, *Software Development Maturity,*
- Chapter 8, *Measurement and Metrics,*
- Chapter 9, *Reuse,*
- Chapter 10, *Software Tools*, and
- Chapter 11, *Software Support.*

Figure 4-12 illustrates how the software engineering management activities discussed here flow into the software life cycle. As you can see, process improvement *[discussed in Chapter 15, Managing Process Improvement]* and risk management are performed continuously throughout the system's life. Consideration of software development maturity and the contractor's commitment to continuous improvement is essential during source selection *[discussed in Chapter 13, Contracting for Success]*. As process improvement succeeds, software development maturity will advance. Once requirements are specified, an architecture can be defined that addresses the system from a domain perspective with regards to the need for open systems *[as discussed in Chapter 3, DoD Software Acquisition Environment]*. The detailed design concentrates on building in quality attributes which include the optimum use of reuse and COTS *[discussed in Chapter 14, Managing Software Development]*. Prototyping and demonstrations *[also discussed in Chapter 14, Managing Software Development]* are used to reduce risk and validate that the design addresses user and technical requirements. Ada should be used as a design language, and of course, it must be used for coding. Models are used throughout the life cycle to define development procedures and analyze metrics data, which are collected throughout. Software engineering tools encompass the entire spectrum of development, and should be used to aid in the implementation of software engineering methods and life cycle activities.

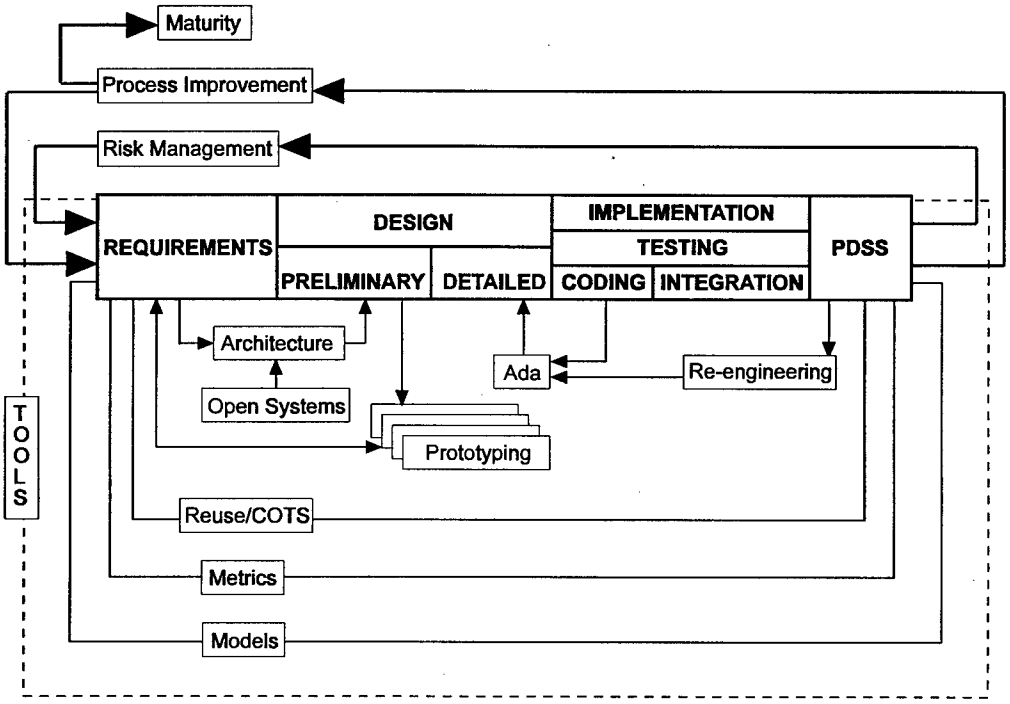


Figure 4-12 Software Engineering Relationship to the Software Life Cycle

Software Engineering Information

An excellent source of information on software engineering is ***CrossTalk***, the monthly publication of the **Software Technology Support Center** *(STSC) [discussed in Chapter 10, Software Tools]. [See box below for information on how to subscribe.]* The need for continued advances in software engineering management and methods is magnified as our reliance on commercial practices and products increases, which also compete in the global marketplace. Timely, topical articles are a means of keeping informed and up-to-date on the latest developments in technology and professional practices amid the momentum of change within the software industry. *CrossTalk* is a high-quality, accurate information link between software managers and practitioners throughout the field. Distributed without charge to DoD and contractor personnel, *CrossTalk* is highly recommended reading for its currency and technical content.

Another periodical, ***Chips***, is a DoD magazine sponsored by the Navy. It has a different focus than *CrossTalk*, covering microcomputer issues (including contracting, networking, software development, policy, training, etc.); however, it occasionally has some software engineering-related articles. *Chips* is distributed free to all government users. Contact the Naval Computer and Telecommunications Area Master Station LANT through e-mail at ***chips@email.chips.navy.mil*** or view *Chips* electronically on the Internet at ***http://www.chips.navy.mil.***

TO SUBSCRIBE: For more information on a free subscription to *CrossTalk*, contact the Software Technology Support Center, Attention: Customer Service, Ogden ALC/TISE, 7278 Fourth Street, Hill AFB, Utah 84056. Phone: (801) 777-8045 or DSN 777-8045, Fax: (801) 777-8069, or DSN 777-8069. E-mail: *custserv@software.hill.af.mil* or Internet at *http://www.stsc.hill.af.mil.* View *CrossTalk* online at: *http://www.stsc.hill.af.mil/www/xtalk.html.*

WHAT IS INFORMATION ENGINEERING?

If you have ever visited the sunny Silicon Valley in California, one of the most popular local curiosities is an enormous house built around the turn of the century by the rifle heiress, Sarah Winchester. As Sarah grew older, she believed she was being haunted by the ghosts of people killed by her husband's rifles. Terrified of meeting these angry souls in the hereafter, she employed two full-time spiritualists who advised her that she would never die as long as her house kept living and changing. For 38 years, the construction of towers, wings, chimneys, rooms, and gardens was nonstop. Sarah's vast fortune was employed to guarantee the constant sound of workers pounding nails, laying cement, digging holes, and chiseling wood. Everything needed to keep the operation-going was on-site — woodshops, cement mixers, warehouses, and supply yards. Because the construction engineers never had a set of overall blueprints showing where the house was going, some rooms were remodeled more than a dozen times. Over the years, throughout this frenzy, oddities began to appear. The house has stairways leading into ceilings, windows blocked by walls, more halls and passages than rooms to connect, a three-story chimney that fails to meet the roof, and many rooms that serve the same purpose.

CHAPTER 4 Engineering Software-Intensive Systems

Like the Winchester Mystery House, the information systems of many large organizations and corporations are under perpetual construction — growing, changing, duplicating, multiplying. There are expanding databases here, new input screens there, spreadsheets everywhere — some systems are changed, updated, and enhanced more than a dozen times. Often vast fortunes are spent keeping these activities going with everything needed to do the job on-site. Over the years oddities begin to appear. The collection of software systems contains masses of unused reports, more bridges and interfaces than systems to connect, data that are inconsistent, redundant, inaccessible, and in incompatible formats, with many systems serving the same purpose. These enormous mystery systems live and change without a set of overall blueprints for the data, systems, and technology needed to support the enterprise. [SPEWAK93]

In the early 1980s, to help stop the constant custom building and replacing of systems with costly odd features, **James Martin** developed the *information engineering* methodology. [MARTIN81] Intending it to contrast with, and complement, software engineering, Martin defined **information engineering** (IE) as,

> *The application of an interlocking set of formal techniques for the planning, analysis, design, and construction of information systems on an enterprise-wide basis across a major sector of the enterprise.* [MARTIN89]

Information engineering is a form of domain engineering oriented towards the MIS domain, which has also proven successful in analyzing C2 systems. IE is predicated on the realization that the procedures for conducting business are in constant flux due to frequent restructuring and changes in organizational focus; whereas, the data requirements of the enterprise are stable. In traditional approaches, database design is dictated by application data requirements developed to automate specific procedures. Every time procedures change, the database must be redesigned. Changing database design to accommodate a change in one procedure has a snowball effect requiring maintenance on all other system components accessing the changed part. Understandably, systems designed this way have extremely high maintenance costs. The goal of IE is to capture the stable data requirements of the enterprise in a database design that remains stable throughout the software life cycle. Dynamic elements are captured in those applications (modules) always subject to change. This process results in substantial maintenance cost

savings. [MICAH90] Because IE focuses first on data rather than on procedures, it is called a ***data-driven*** method.

Information Engineering Process

According to Finkelstein, Martin's co-author, the IE process is characterized by two distinct stages: a technology-independent and a technology-dependent stage, as illustrated in Figure 4-13. The starting point is strategic business planning which allows for continual evaluation and refinement of the Strategic Plan at all stages of development, as illustrated on Figure 4-14. Using this method, feedback is quick, exact, and effective, with clear communication links and precise implementation. [FINKELSTEIN92]

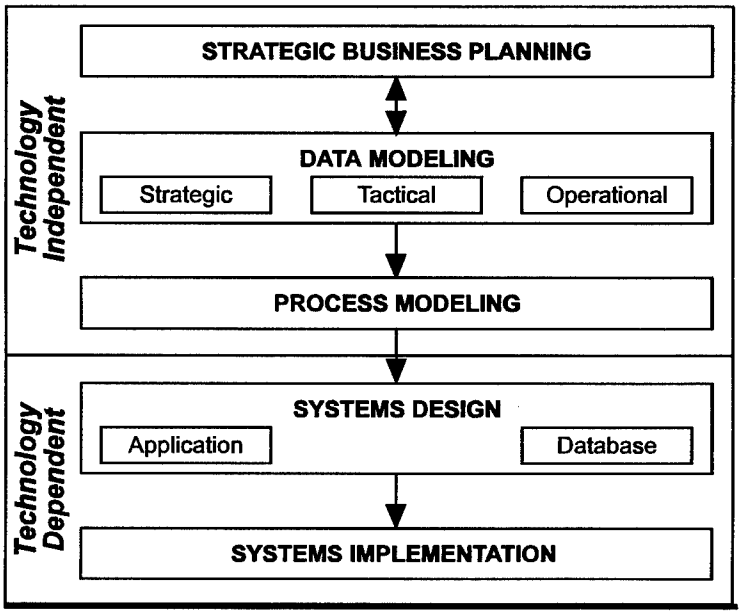


Figure 4-13 Information Engineering Phases [FINKELSTEIN92]

Information Engineering Architecture

IE addresses four architectural levels which separate data and process, thus creating databases and applications that are flexible and facilitate rapid changes and enhancements in response to competitive pressures. These levels are illustrated on Figure 4-15.

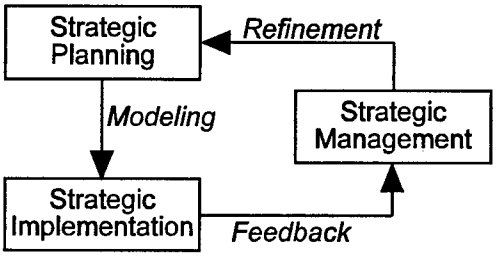


Figure 4-14 Strategic Management Planning
[FINKELSTEIN92]

	DATA	PROCESS		
Corporate Vision	Planning Statements	Business Events	*Business Plan*	*Business Level*
Technology-Independent	Data Model	Process Model	*Business Model*	*Logical Level*
Technology-Dependent	Database Design	Application Design	*System Model(s)*	*Physical Level*
Site-Dependent	Database	Application Code	*Implementation(s)*	*Platform Level*

Figure 4-15 Information Engineering Four-Level Architecture
[FINKELSTEIN92]

- **Level 1: Business**. This reflects the corporate vision because data based on strategic planning statements are defined at all management levels. The business plans operate on these data, based on planning statements and business events which process the data.
- **Level 2: Logical**. Planning statements based on the corporate vision are used to develop technology-independent data models. Process models, represented by the business model, are developed from data models and business events based on the Business Plan.
- **Level 3: Physical**. Technology-dependent database designs are developed based on the data and process models. These database designs and relevant process models (representing the system models) provide input to application design (which also feeds back to database design).
- **Level 4: Platform**. The database design is physically implemented as site-dependent databases. Application code operating against

can be implemented and applications executed on specific platforms that employ the best available hardware, software, and communications technologies. [FINKELSTEIN92]

NOTE: Make sure someone in your program office understands IE well enough to interpret and review contractor IE effort products. Otherwise, there is a the risk that the stacks of paper produced by this process will be incorrect or ignored.

IE is discussed as an example of one method for modeling data. Other methods for MIS are also viable, such as essential systems analysis [McMENAMIN84], object-oriented analysis *[see Chapter 14, Managing Software Development]* [COAD90], and IDEF *[described next]*. The approach you adopt requires research and an understanding of your program to determine which is most applicable to your program-specific needs.

IDEF

Integrated Computer-Aided Manufacturing Definition Language (IDEF) is a modeling technique that supports IE. It was initially developed in the 1970s for Air Force Logistics Center support programs in the manufacturing environment. In 1989, an IDEF users' group was formed to establish a methodology for implementing the IDEF approach which provides a framework for classifying important information about an enterprise. ***The main goal of the IDEF exercise is to identify areas for process improvement.*** Improvements can be in the areas of:

- Manual procedures and techniques,
- Product and service quality,
- Industrial processes and factory automation,
- Information systems and computer automation,
- System development methods, and
- Business procedures, to name a few.

IDEF activity modeling captures and graphically depicts the specific steps, operations, and data elements needed to perform an enterprise activity. An **activity** is defined as a named process, function, or task that occurs over time and has recognizable results. As illustrated in Figure 4-16, each activity is represented by a rectangle. Entering, exiting, or linking activities are those factors that change the activity. These fall into the categories of:

- **Input data** or material for the activity (e.g., program requirements),
- **Controls** that regulate the activity (e.g., engineering principles, existing policies),
- **Output data** or materials produced by the activity (e.g., quality software), and
- **Mechanisms** comprised of people or machines that perform the activity (e.g., new technology).

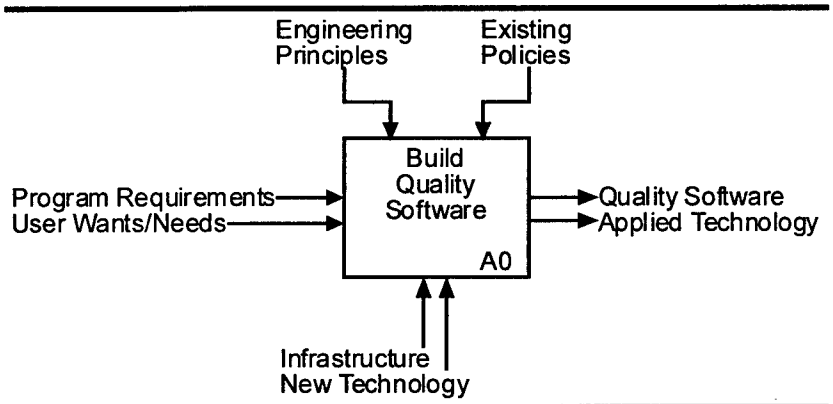


Figure 4-16 IDEF (Level A0) for Nominal Program

The interrelationships among activities are modeled by using node trees, as illustrated on Figure 4-17 (below). An activity can be decomposed into subactivities which can be further decomposed into sub-subactivities. **Context diagrams** and **decomposition diagrams** are used to provide both overall and more detailed breakdowns of activities. In a typical program, the scope and requirements are defined first. Then the information required to support the activities is gathered through a series of working sessions that include users and systems experts. These data are finally captured in an automated tool for documentation.

The IDEF approach is recommended ***before*** starting new MIS programs. It aids in mission area analysis, functional analysis, and the strategic planning. IDEF can be also used to model *alternative views* of the enterprise. The IDEF approach uses hierarchically decomposed function models and **entity-relationship (E-R) diagrams**. *[An E-R diagram identifies data objects and their relationships through a graphical notation.]* These views then become the baseline upon which to plan and implement process improvement. The IDEF modeling approach leads to an understanding of the total enterprise

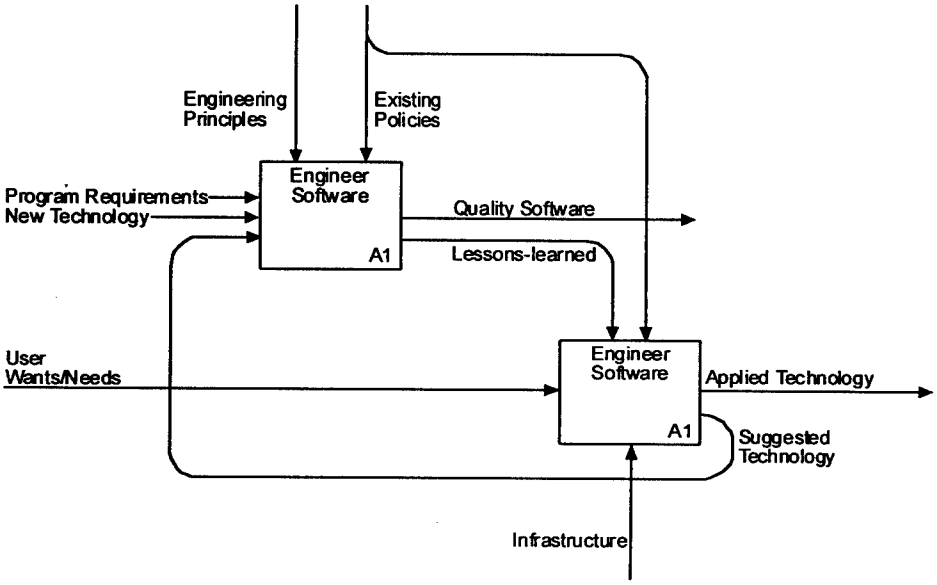


Figure 4-17 IDEF (Decomposition of A0) Model for a Nominal Program

by integrating business tasks, rules, and objectives to work together in a productive way. *[See Volume 2, Appendix A for a point of contact for IDEF information.]*

SUCCESS THROUGH ENGINEERING

Mosemann summarized why engineering is the solution for successful software-intensive systems acquisition and management when he emphasized that,

> *...we've got to adopt an engineering focus. We have got to concentrate on cost-effective solutions, solutions that are built from models, and on using capable, defined processes, rather than focusing on perfect systems that meet 100% of our wishes. Again, this is a management challenge, not a technical challenge. There's just no way to manage or to control the configuration, to control the side-effects, in these kinds of large software developments unless we use engineering discipline.* [MOSEMANN92[1]]

As illustrated on Figure 4-18, software engineering requires more emphasis and resources upfront during the development effort. This change from a traditional software-as-art approach (where most of the resources are spent in the support phase) to a software engineering approach reduces the total amount of resources necessary, since the resultant software support costs are substantially reduced. Well-engineered software lays a solid foundation for the system to evolve into its operational environment by employing sound development practices and procedures. Through the software engineering discipline you will have available to you:

- Comprehensive methods for all software development phases,
- Better tools for automating these methods,
- More powerful building blocks for software implementation, and
- An overriding philosophy for coordination, control, and management. [PRESSMAN92]

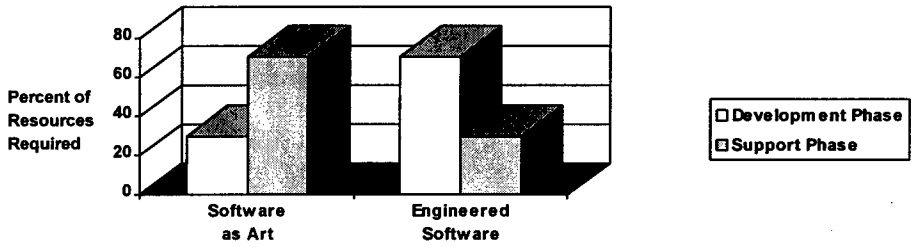


Figure 4-18 Software Engineering Builds a Solid Foundation Upfront

The challenge is to acquire and develop systems that meet the user's needs given the usual performance, life cycle cost, and schedule criteria. However, the only way to meet this challenge and achieve acquisition success is to use engineering discipline in all aspects of software development. Anything less will only produce schedule slips, cost overruns, and systems that do not meet user needs.

REFERENCES

[Ada/C++91] *Ada and C++: A Business Case Analysis*, Office of the Deputy Assistant Secretary of the Air Force, Washington, DC, June 1991

[BOOCH94] Booch, Grady and Doug Bryan, Software Engineering with Ada, Third Edition, Benjamin/Cummings Publishing Company, Redwood City, California, 1994

[BULAT95] Bulat, Brian G., "Space & Warning Systems Center Domain Engineering Experiences," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, 1995

[CARDS92] *Acquisition Handbook Central Archive for Reusable Defense Software (CARDS): Informal Technical Data*, STARS-AC-04105/001/00, Electronic Systems Center, Hanscom AFB, Massachusetts, September 4, 1992

[COAD90] Coad, Peter and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[DSMC90] Systems Engineering Management Guide, Defense Systems Management College, U.S. Government Printing Office, Washington, DC, 1990

[EIA632] Electronic Industry Association (EIA), Interim Standard 632, ((draft)), *Systems Engineering*, September 20, 1994

[FINKELSTEIN92] Finkelstein, Clive B., "Information Engineering: Strategic Systems Development," Jessica Keyes, editor, Software Engineering Productivity Handbook, Windcrest/McGraw-Hill, New York, 1992

[HOLIBAUGH92] Holibaugh, Robert, "STARS Domain Analysis Survey" briefing, Software Engineering Institute, June 15, 1992

[HUMPHREY89] Humphrey, Watts S., Managing the Software Process, Software Engineering Institute, Addison-Wesley Publishing Company, 1990

[KERSH90] Kersh, Pvt Gerald, as quoted by Robert A. Fitton, ed., Leadership: Quotations from the Military Tradition, Westview Press, Bolder, Colorado, 1990

[KOGUT94] Kogut, Paul, Kurt Wallnau, and Fred Maymir-Ducharme, "Software Architecture and Reuse," TriAda '94, Baltimore, MD

[LYONS92] Lyons, Lt Col Robert, as quoted by David Hughes, "Digital Automates F-22 Software Development with Comprehensive Computerized Network," *Aviation Week & Space Technology*, February 10, 1992

[MARTIN81] Martin, James, and Clive B. Finkelstein, Information Engineering, Savant Institute, Carnforth, Lancs, United Kingdom, 1981

[MARTIN89] Martin, James, Information Engineering, Book 1 (of 3): Introduction, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989

[MAYMIR95] Maymir-Ducharme, Fred and David Weisman, "US Air Force Comprehensive Approach to Reusable Defense Software (CARDS) Technology Transition Program: Reuse Partnerships," Reuse '95, Morgantown WV 1995

[McGRAW89] Dictionary of Scientific and Technical Terms, Fourth Edition, Sybil P. Parker, Editor-in-Chief, McGraw-Hill Book Company, New York, 1989

[McMENAMIN84] McMenamin, Steve and John Palmer, Essential Systems Analysis, Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey, 1984

[MICAH90] "Micah Systems, Inc. and Information Engineering," October 24, 1990

[MOSEMANN91] Mosemann, Lloyd K., II, "Air Force Software in the Year 2000," keynote luncheon address, STSC-HQ USAF/SC Joint Software Conference, Salt Lake City, Utah, April 16, 1991

[MOSEMANN92[1]] Mosemann, Lloyd K., II, "Comments on MIL-STD-499B," October 23, 1992

[MOSEMANN92[2]] Mosemann, Lloyd K., II, "Software Management," keynote closing address, Fourth Annual Software Technology Conference, Salt Lake City, Utah, April 16, 1992

[MOSEMANN92[3]] Mosemann, Lloyd K., II, "Software Measurement and Quality," keynote address, Fourth Annual REVIC User's Group Conference, Fairfax, Virginia, March 25, 1992

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[SHINA91] Shina, Sammy G., "Concurrent Engineering," *IEEE Spectrum*, July 1991

[SPEWAK93] Spewak, Steven H., with Steven C. Hill, Enterprise Architecture Planning: Developing a Blueprint for Data, Applications, and Technology, QED Publishing Group, Boston, 1993

[STRASSMANN91] Strassmann, Paul A., as quoted by Bob Brewin, "Corporate Information Management White Paper," *Federal Computer Week*, September 1991

[STRASSMANN92] Strassmann, Paul A., "Joining Forces to Engineer Success: The DoD Context," opening keynote address, Fourth Annual Software Technology Conference, April 14, 1992

[WAGNER95] Wagner, Capt Gary F., and Capt Randall L. White, "F-22 Program Integrated Product Development Teams: How One Major Aircraft Program Developed Integrated vs. Independent Product Teams," *Program Manager*, Defense Systems Management College Press, July-August 1995

[WANG86] Wang, An, as quoted in *Boston Magazine*, December 1986

[ZELLS92] Zells, Lois, "Learn from Japanese TQM Applications to Software Engineering," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[ZRAKET92] Zraket, Charles E., "Software Productivity Puzzles, Policy Changes," John A. Alic, ed., Beyond Spin-off: Military and Commercial Technologies in a Changing World, Harvard Business School Press, Boston, Massachusetts, 1992

CHAPTER 4
Addendum A

SWSC Domain Engineering Lessons-Learned

As discussed in Chapter 9, *Reuse*, the **Space and Warning Systems Center (SWSC)** domain engineering team presented their lessons-learned from two years experience on the **Space Command and Control Architectural Infrastructure (SCAI)** re-engineering program in April 1995 at the Software Technology Conference, Salt Lake City, Utah. The SWSC at Cheyenne Mountain, Colorado, maintains and modifies C2 systems for US. Space Command, North American Aerospace Defense Command (NORAD), and Air Force Space Command (AFSPC). These systems are comprised of 26 stovepipe systems, 12 million lines-of-code, 24 different languages, 34 separate operating systems, and numerous proprietary hardware and software components — all having complicated software support environments, as illustrated on Figure 4-19 (below). This maintainer's nightmare was fertile ground for architecture-based domain engineering.

The SCAI Project is using the domain engineering approach developed by the **Software Technology for Adaptable and Reliable Systems (STARS)** program called ***megaprogramming*** *[discussed in Chapter 9, Reuse].* They used domain analysis to create a domain-specific architecture to which all individual systems in the domain are mapped to ensure product-line software reuse, as illustrated on Figure 4-20 (below).

The architecture developed for the SCAI program is decomposed into a layered domain requirements model (DRM) and a set of application architectural models (AAM)s. The current scope of the DRM is the SWSC space domain; each AAM is specific to one system in the domain. The layered DRM is a modified Booch-type model, while the AAM is a network topology model and a mapping of application

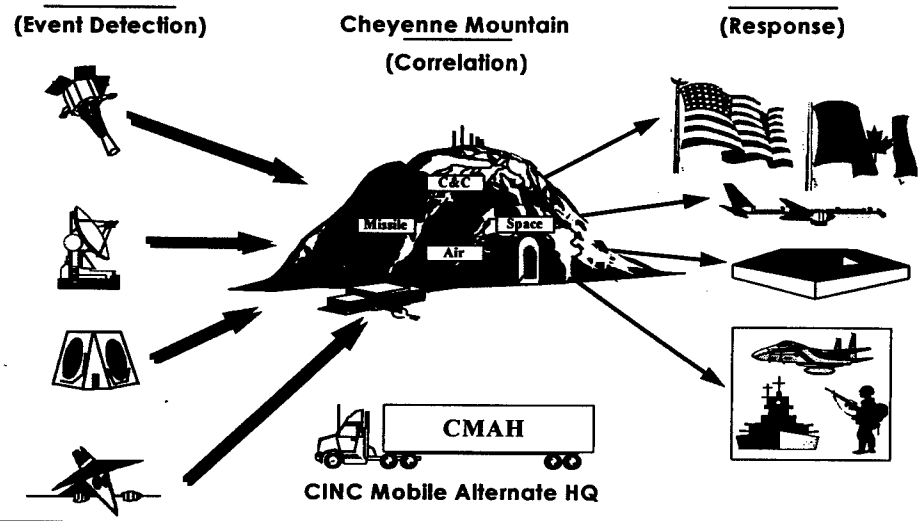


Figure 4-19 SWSC Software Re-engineering Environment [BULAT95]

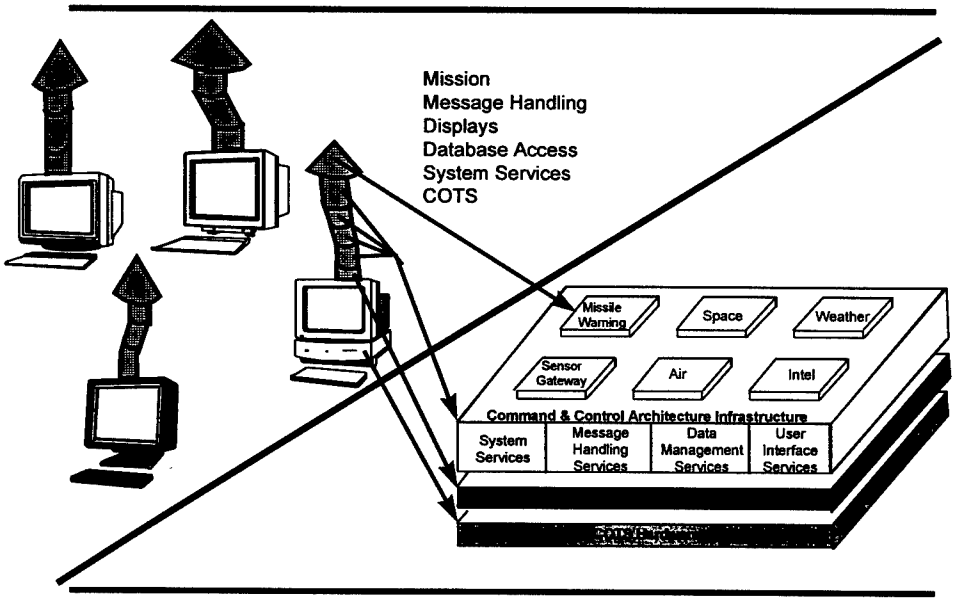


Figure 4-20 SWSC Domain Engineering Approach [BULAT95]

tasks to machines. [BOOCH94] These models comprise the SCAI domain architecture, as illustrated in Figure 4-21.

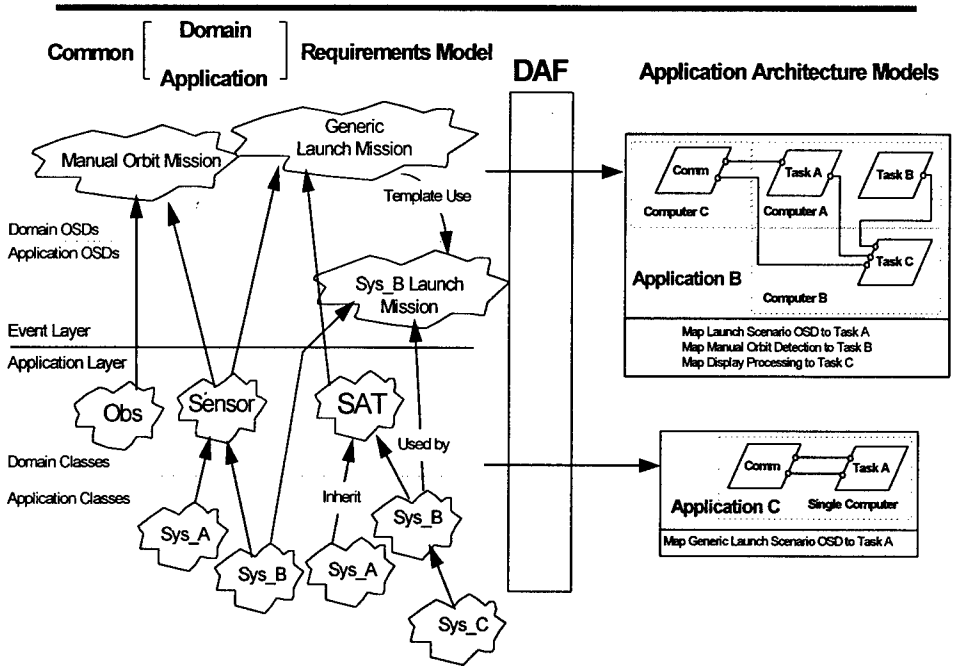


Figure 4-21 Domain/Application Model Relationship [BULAT95]

The SCAI is using a product-line software development approach which implements two software life cycles: one life cycle where generalized domain products are developed, and a parallel life cycle where individual applications are constructed from domain life cycle products, as illustrated on Figure 4-22 (below). Obviously, domain products must exist prior to their use in the application. The problems the team encountered associated with the requirement for prior construction of all domain components were:

- There were substantial upfront domain engineering costs not associated with developing any product. (As DoD shrinks, these costs are increasingly difficult to justify.)
- Generalized models and generalized components can only be validated through their use on real systems. Even within a single system, a reusable class must be validated in each of the contexts in which it is used. Monolithic *waterfall* systems development has

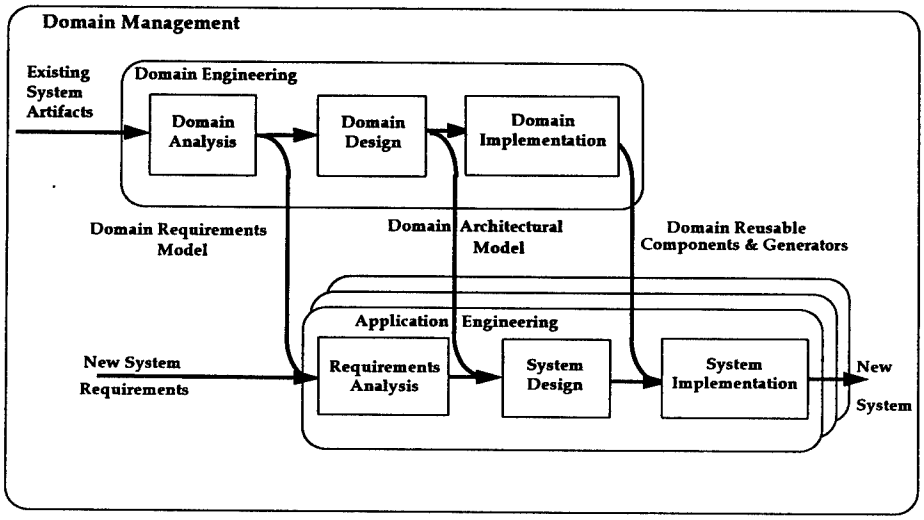


Figure 4-22 Iterative Two Life Cycle Domain/Application Engineering Process [BULAT95]

largely been discredited *vis-à-vis* more iterative approaches to modeling and systems development.

- A large domain, such as C2, may contain many complex systems. In spite of the fact that all these systems have much in common, it is unlikely that domain engineering can be initially accomplished within the scope of every one of these systems before the need to deliver the first re-architected system occurs.

For the above reasons, the domain engineering process must allow for the iterative acquisition of domain knowledge. The SCAI architecture framework was constructed from processes which are inherently iterative; therefore, the overall process is iterative. As new systems are analyzed and the scope of the DRM is extended, more domain missions are identified, new classes are created, and existing classes are generalized. Newly generalized classes are then reinserted into existing missions and retested.

SWSC's goal is to demonstrate that domain engineering will increase software quality, while decreasing the cost of developing and maintaining families of related SWSC C2 systems. As of January 1995, the first intermediate delivery of the SCAI system (the service layer) indicated that over 50% of the code was reused or had been generated. Table 4-1 illustrates the program costs for each release (both actual and adjusted) and the amount of program code delivered

at the end of each release. It shows the incremental cost per line-of-code, the cumulative program cost, and the cumulative cost per line for each release. The last column show how initial costs are amortized. As more and more code is developed using the megaprogramming approach, it becomes increasingly cheaper.

SCAI	PROGRAM EXPENDI-TURES ($Millions)	ADJUSTED COST* ($Millions)	INCRE-MENTAL LOC	INCRE-MENTAL COST/LOC	CUMU-LATIVE COST ($Millions)	CUMU-LATIVE COST/LOC
Pilot	$2.9	$14.0	125K	$112.0	$14.0	$112.0
Release 1	$3.4	$8.5	267K	$32.0	$22.5	$57.0
Release 2**	$2.6	$6.5	230K	$28.0	$29.0	$47.0
Release 3**	$2.0	$.50	150K	$33.0	$34.0	$44.0
TOTALS	$10.0	$34.0	772k			

*Values adjusted to full life cycle cost plus prior domain work
**Estimated

Table 4-1 SCAI Cost with Megaprogramming

CHAPTER 4
Addendum B

Software Reliability: A New Software OT&E Methodology

Captain Randy McCanne
Software Test Manager
HQ AFOTEC/SAS

NOTE: This article is found in Volume 2, Appendix O, Additional Volume 1 Addenda.

CHAPTER

5

Ada: The Enabling Technology

CHAPTER OVERVIEW

No other software language complements and facilitates the software engineering process better than Ada. Ada combines, in a coherent manner, the best features of many previous languages used by DoD to remove errors early, to improve software reliability, to permit software reuse, and to simplify system maintenance. As ***Mosemann*** *explains,*

> *We use Ada because Ada facilitates software engineering and we need engineered software. But let me suggest it is not only military software that must be engineered. Management information systems, process control, telecommunications — every large scale commercial application of which I can conceive requires the robustness and complexity control of engineering techniques. The military requirement for engineered software is not unique. And since Ada is, for the foreseeable future, the language for software engineering, a strong and wide-ranging Ada capability is vital to the Air Force, to the US Defense industrial base, and the country as a whole.*
> [MOSEMANN89]

Although initially developed by DoD for military applications, Ada offers advantages for many other application domains due to the methods its developers used to arrive at the final definition of the language. [NATO86] The latest version, Ada 95, embodies object-oriented programming and is suited for all application domains. In this chapter you will learn why Ada's features make it the best language choice where safety and reliability are paramount, how Ada enables the principles of software engineering, the advantage's of Ada's use, technology considerations, new features found in

CHAPTER 5 Ada: The Enabling Technology

Ada 95 (an ISO standard and the world's first internationally standardized object-oriented programming language), and development and management pitfalls to avoid. Ada implementation is discussed in light of whether your program is a new start, on-going, or you are responsible for the maintenance of legacy software-intensive systems.

Ada is a language designed to incorporate the principles of software engineering that enable the fulfillment of software engineering goals [discussed in Chapter 4, Engineering Software-Intensive Systems]. Ada contains a full set of control structures and statements to produce reliable software while reducing life cycle costs. It differs from other languages in its ability to define types and subprograms, and by its strong support of tasking, real-time and systems programming, and packaging. Ada accomplishes system-level programming by enforcing strong control over data representation and access to system-dependent properties. Ada packages are collections of logically-related entities (types, data, subprograms, and tasks). Ada enables modularity by allowing data, types, and subprograms to be packaged and compiled separately. This leads to physical modularity that permits independent development and testing of small packages before they are integrated into the larger application. The Ada package is made up of two parts: the specification (information visible to the user and other units) and the body (details invisible to the user and other units). This separation of specification and body within the package enables logical modularity and allows other units to access only the logical properties (not the implementation details) of the package. [ALLEN92] All these features enable software that is reliable, supportable, and reusable.

There are several technology issues to consider if you are developing a new system or upgrading an old one. These include compiler maturity, run-time efficiency, support tools, impacts on requirements and design, porting to other resources, and re-engineering [discussed in Chapter 11, Software Support]. Selecting a compiler is a serious management decision to be addressed in your ***Risk Management Plan****. You need to select a mature, validated compiler that fulfills your program-specific requirements. You should use support services and available benchmarks to help in your selection. Ada applications must interface with other systems; therefore, development of open system environments are essential. Operating systems, databases, graphics, and windowing environment technology issues demanding your attention are covered here.*

CHAPTER
5

Ada: The Enabling Technology

Ada: BECAUSE IT'S SAFE

One of the main design goals in developing the Ada language was to facilitate safety in weapon systems. **Lieutenant General Robert Ludwig** (USAF) underscored this when he stated:

> *Ada is the language of choice where human life is at stake. Sometimes software people spend so much time staring at their computer screens, they forget that our fellow warriors are strapping on your software and putting their lives on the line. When human life is at stake, the question is not what language is the easiest to use or the most popular, it is what language will give us the highest safety and reliability. NASA, the FAA, commercial airliner companies, and many others all chose Ada for the same reason.* [LUDWIG92[1]]

Ada's Eagle — Our Safest Bird of Prey

If you have to go to war today, there is probably no better place to fight it than from the cockpit of an **F-15 Strike Eagle**, touted as being *"one of the most capable, most cost-effective, and safest aircraft now available."* With an air-to-air combat record of 96.5 wins to zero losses, and with a loss rate from all other causes (excluding combat) of approximately 2.6 per 100,000 flight hours, the Eagle is the safest fighter in the US inventory. [AW&ST94] This version of the F-15 is referred to as ***"Ada's Eagle"*** by the joint program team who built the **Ada-based Integrated Control System (ABICS)** for her athletic airframe. *Beefed-up* and *souped-up*, Ada's Eagle can

turn-and-burn with the best of them. In addition to a quick reaction capability, the Ada-based integrated flight and fire control technology allows for highly lethal *air-to-air* gunnery prowess, in addition to mighty *air-to-surface* muscle (the Strike Eagle can unleash up to 24,500 pounds of ordnance with an accuracy that rivals any purebred bomber.) [DANE92]

The new F-15E has approximately 2.3 million lines-of-code, a good portion of which enables the weapon systems officer (WSO) *[called the "wizzo"]* to easily select targets, plot intercepts, and choose munitions, allowing the pilot to perform his mission undistracted. Equipped with **LANTIRN** night attack systems and **APG-7 radar**, the Eagle can locate a vehicle in a jungle or select and hit the door of a house in the middle of town. [AW&ST90] The crew can perform their mission with confidence and accuracy because Ada is governing their aircraft's mission-critical and flight-critical systems which manage its flight controls, fire controls, navigation, and sensor systems. As **Lt Gen Ludwig** remarked, "*It is truly amazing that today two people in that F-15E can accomplish in one sortie what it took a thousand men in 100 bombers, with many lost lives on every mission, to do in World War II.*"

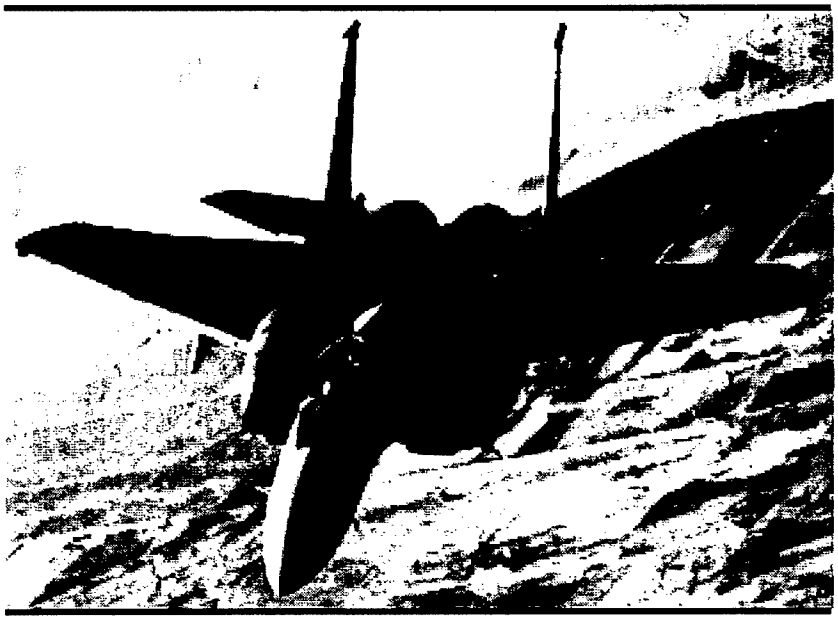


Figure 5-1 Ada's Eagle: Our Safest Fighter

CHAPTER 5 Ada: The Enabling Technology

NOTE: See Chapter 1, *Software Acquisition Overview*, for more Ada success stories about the F-22 and Boeing 777.

Ada: BECAUSE IT'S SMART

Saving lives is one of the many end-products of why we use Ada. How we achieve that product is another reason why Ada is the language of choice for DoD. **General Ronald Yates** (USAF retired) defined Ada as "*more than a common language. It is an enabling technology for software engineering.*" [YATES91] The **Software Engineering Institute (SEI)** states:

> *Ada is unlike other languages, however, in the degree to which it fosters and supports the practice of software engineering principles. These design principles are believed to lower software development costs, increase software quality, and lower maintenance costs, especially for large or complex systems. In effect, these features and the structure of the language make it easier to develop software that is more understandable and more maintainable.* [SEI90]

Ada Is for All Domains

As technologies have evolved, the similarities among weapon system, C3, and MIS domains have outweighed any differences, making Ada a *best practice* for all. Although originally developed for real-time weapon systems applications, Ada has been used in developing MISs in both the government and private sectors since the first Ada compilers became available. In fact, the new version of Ada, **Ada 95,** includes advanced features for enhancing the Ada MIS environment.

Another indicator of why Ada is not just for weapons systems is the phenomenal growth in its use within the **commercial sector**. The US market for Ada hardware and software products was $975 million in 1989. If current growth trends persist (20% increase per year), ***by 1996, the market is expected to reach between $1.8 and $2.9 billion***. [DIKEL91] This growth in Ada use has occurred because major corporations are increasingly embracing Ada where safety and reliability are their bread and butter. The majority of Ada's non-weapon system applications are for core business capabilities, such as

manufacturing process control, industrial design, telecommunications, and diagnostic analysis. Ada is used in a wide spectrum of industries, such as transportation, finance, health care, energy, and national security. *[See the section below "Ada: Who's Using It?" for examples.]*

While Ada's original focus was to be a single, flexible (yet portable) language for real-time embedded systems, *its application domain has expanded* to include many other areas, such as large-scale information systems, distributed systems, scientific computation, systems programming, telecommunications, process control and monitoring systems. [INTERMETRICS95] As with embedded weapon systems, Ada is the language of choice for other application domains where safety and reliability are paramount and a system abort is intolerable. Examples of large-scale **Ada MIS systems** abound throughout the services. The Marine Corps has developed many MISs to include: paycheck systems (and upgrades), bar code scanner systems, materiel management systems, supply accounting systems, financial disbursement systems, and inventory tracking systems. The Army, requiring Ada for all software developments, used Ada to build two major MISs: the **Standard Installation/ Division Personnel System-3 (SIDPERS-3)** and the **Standard Finance System Redesign (STANFINS-5R).** The Air Force has numerous program management systems, a word processor, training systems, data manipulation systems, accounting systems, and other financial information systems — all in Ada. [REED89]

It is to your advantage to realize that ***Ada is the language of choice*** for military and commercial software engineers who are building mission-critical, safety-critical, or security-critical systems that demand sturdy, supportable, portable, and transparent software. The DoD's interest in engineered software translates directly into its interest in Ada. [GROSS92]

Ada's Background

The main issue with higher order languages (HOLs) has been standardization. When HOLs first became popular, hundreds of languages were developed and supported — many for specialized applications on specific computers. In DoD alone, some 400 different HOLs were used. Many of these languages were not well-suited for mission critical systems and other large military applications. Also, they were not controlled, so dialects proliferated. [NATO86] ***A standardized language became a necessity.***

Standardization. Standardization of a language implies a stringent definition for **syntax** (or grammar; i.e., the way terms are combined) and **semantics** (what the terms mean). Any changes to the language must be rigorously controlled. The language's compilers must be standardized, since any compiler deviation becomes a *de facto* language deviation. A standard language must also be *portable* (i.e., usable on different computers without modification). Agencies involved in language standardization include: the **International Standards Organization (ISO)**, the **American National Standards Institute (ANSI)**, the **Institute of Electrical and Electronic Engineers (IEEE)**, and **DoD**. DoD's solution to the standardization problem was finally reached on January 22, 1983 when ANSI/MIL-STD-1815A, *Ada Programming Language*, was put into effect—marking the ***standardization of the Ada language which was developed by DoD to standardize software development and support.***

A standardized language had three benefits for DoD. First, software personnel in DoD and its contractor community had become fragmented with the proliferation of languages used for military software prior to 1983. This meant software professionals were very specialized and could not move readily from one program to another. Second, so many languages meant that DoD had difficulty in transporting software across computer environments. In addition to the cost of rehosting software, the diversity of development languages meant DoD could not easily reuse what it had as a capital investment of pre-developed, pretested legacy software components. Finally, the large number of languages meant that few commercial software tools were available for any given language. With the consolidation of the considerably large DoD software market into one language, tool makers would have a larger, single market upon which to concentrate their efforts. Thus, they would have the incentive to produce more and better tools competitively priced for DoD. To summarize, DoD anticipated significant savings in personnel, training, software reuse, and tools by consolidating its software into one standardized language — Ada. [SEI90]

It is normal practice for ANSI to automatically review a standard every 5 to 10 years for continued applicability. In 1988, the decision was made to revise Ada 83 and update the ANSI and ISO standards. [INTERMETRICS95] The culmination of this revision occurred on February 15, 1995, when Ada 95 became an ISO standard (ISO/IEC 8652:1995), making ***Ada the world's first internationally-standardized object-oriented programming language!***

An example of the benefits achieved by using a standardized language is the re-engineering of the 20-year old **Standard Army Management Information Systems (STAMIS)** to run in Ada. A pilot re-engineering program was conducted which included a part of the STAMIS called the **Installation Materiel Condition Status Reporting System (IMCSRS)**, which was written in COBOL. By using a standardized language, the Army was able to bypass coding many of the modules by using components from the **Reusable Ada Products for Information Systems Development (RAPID)** library *[now the **Defense Software Repository System (DSRS)**] [see Chapter 9, Reuse].* They were able to submit reusable generic modules developed for STAMIS to the library for use in future DoD programs. Programmer work loads were also drastically reduced — two programmers with no previous Ada experience built the IMCSRS in a record 4½ months.

In an interview with *Government Computer News*, **Emmett Paige, Jr.**, Assistant Secretary of Defense for C3I, reasserted his support of Ada. He stated, "*I think the problems that brought us to Ada still exist.*" He emphasized that the real cost of software-intensive systems lies in supporting the system after it is deployed, not in its development. "*That's where Ada was supposed to help us,*" he said, "*and I believe it has.*" [PAIGE93]

Ada: BECAUSE IT ENABLES SOFTWARE ENGINEERING

The Ada language was developed to support software engineering goals through built-in features that draw on the software engineering principles. However, using Ada alone will not automatically result in the production of well-engineered software. It must be used in conjunction with the application of software engineering principles. As **Lieutenant General Albert J. Edmonds** (USAF), while Director for Command, Control, Communications, and Computer Systems, Joint Chiefs of Staff, stated, the **F-22** Dem/Val program "*wasn't a success only because of the selection of Ada as the programming language. It was a success because leadership was committed to a disciplined approach to software development.*" [EDMONDS93[2]] If you thoroughly understand the principles of software engineering and apply them to your program, the use of Ada can help you effectively achieve the software engineering goals.

CHAPTER 5 Ada: The Enabling Technology

NOTE: See Chapter 4, *Engineering Software-Intensive Systems*, for a discussion on software engineering goals and principles.

Software Engineering Principles and Ada

Ada was developed to directly support the principles of software engineering *[discussed in Chapter 4, Engineering Software-Intensive Systems].* It aids in the development of software solutions that are modifiable, reliable, efficient, and understandable. In an interview with *Government Computer News*, **Belkis Leong-Hong**, acting director of DISA's Center for Information Management, explained how Ada relates to software engineering.

> *If you think back to the time when Ada was first put forth, it was a very valiant effort on the part of DoD to say, what we really have is a software development discipline...that says if you adhere to the software engineering discipline upfront, ultimately the cost of maintaining software is going to be significantly lower. Everything is predicated on the fact that you're going to do things right at the beginning. You're going to have traceable code. You're going to have modularity. All of these are good things that the principles of software engineering are founded upon.* [LEONG-HONG93]

Ada: The Great Facilitator

The principle advantages of Ada are in the way it ***facilitates software engineering***, particularly in the areas of software design and implementation. Ada provides facilities for reducing two major problems associated with large scale software developments: poor component interfaces and interference between components sharing global data.

Multiple component interfaces. Multiple interfaces between separate software components often lead to the production of incompatible components by different programmers. Ada requires that interfaces between separate components of a software application be precisely defined to ensure the compatibility of components produced by different programmers. This is accomplished by enforcing the same interface definition for all modules via the specification.

This differs from C header files, because the interfaces defined in a C header file need not be used by module users.

The **YF-22 prototype development** was an excellent example of how using Ada solves interface problems. Software for the aircraft (comprising 35% of the total avionics development cost) was developed by eight geographically-separated subcontractors. All the Ada code, including ground systems and test mock-ups, were written using different Ada compilers and hardware platforms. The 12 major avionics subsystems, such as flight systems monitoring, navigation, weapons control, and defense systems, comprised over 650 Ada modules.

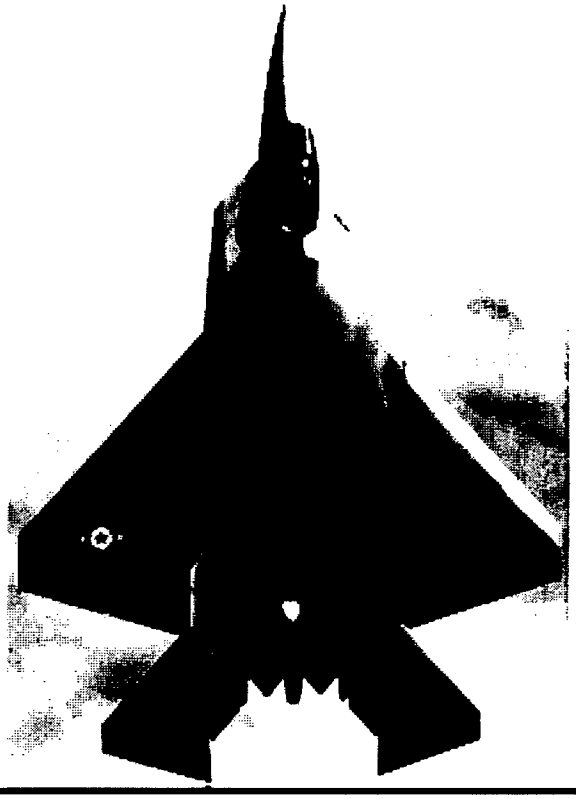


Figure 5-2 Ada Component Interfacing Facilitated FY-22 Fire Control Software Demonstration

The eight teams brought their software together at least six times for major demonstrations, such as the fire control software demonstration. As explained by **Lieutenant General Edmonds**,

> *Picture this. 12 subsystems, 650 modules, and millions of lines-of-code...stack 'em up...And listen to this part: how long do you think it took to integrate this job? All this software. A year? Six months? No. Six weeks? Think again: 650 modules, 12 subsystems, and millions of lines-of-code — a lot of code, and a lot of "stuff," and a lot of places — and a lot of states. ...How long do you think it took to integrate this software? If you guessed three days, you're right. Three days. **T, H, R, E, E** days!* [EDMONDS93[1]]

Key to integration success were the shared Ada package specifications with their enforced interfaces. ***No other language could have supported this level of distributed development on such a large, complex program.*** "*Nothing*," Edmonds exclaimed, "*can compare with its [Ada's] ability to integrate different efforts.*" [EDMONDS93[2]]

Interference between components sharing global data. **Interference** can be created between components written by different programmers using global (shared) data incorrectly modified by various program units. Ada's **packaging feature** helps reduce the need for global data, and through its **data hiding** capabilities Ada minimizes the possibility of interference. Ada also reduces problems because it is a strictly **controlled standard** for which its standard was issued before proliferation of compilers. Each Ada **compiler** must be periodically validated (usually annually) against an extensive set of tests to check conformance with the Ada standard. This procedure ensures software developed in Ada, for any given system, can be reused in other systems without the need for significant re-development. It also ensures that it will interface with other Ada software, regardless of the Ada software engineering environment (SEE). This results in reduced development time, higher productivity, and significant cost savings.

In contrast, 18 different companies develop and market C++ compilers and they all call them "*C++*" — however, they all operate on proprietary versions of the language. C++ written with one company's compiler is not necessarily portable to another C++ software environment.

Ada: FACTS AND FALLACIES

According to **Don Reifer**, former Chief of the **Ada Joint Program Office (AJPO)**, the basic reason you hear about problems surrounding Ada is ***popularity***. Old habits are hard to break. The AJPO has had to counter every argument against Ada: pricey tools, no training, no bindings, too hard, etc. The following is a list of common Ada fallacies and the facts that negate them.

Fallacy: ***No one is offering any courses in Ada.***
Fact: There are 302 universities offering Ada as its primary course on programming languages. There are 550 Ada courses being offered this year to fulfill curriculum requirements in Computer Science I and II. Both the Air Force Academy and West Point are teaching Ada in their computer science departments. The Air Force Institute of Technology is solely Ada for their computer science and software engineering courses. At the Naval Academy, Ada courses are mandatory for all computer science graduates, and at the Naval Postgraduate School (a leader in Ada software engineering) Ada is the initial programming language taught. *[More about Ada training and a wealth of other information (including FREE tutorials) is available from on the Ada Information Clearinghouse (AdaIC) Web server. See Volume 2, Appendix B for the AdaIC's Web address.]*

Fallacy: ***Ada is too hard to learn.***
Fact: Ada is not too hard to learn, especially for those with an understanding of software engineering principles. If that is not the case, do you really want them developing your software? Addendum A gives estimates on the length of time to learn Ada. In addition, there are many educational resources available.

Fallacy: ***Ada will force me to abandon my existing libraries.***
Fact: Ada 95 defines set interfaces with C, COBOL, and Fortran. Ada programmers can invoke libraries of these languages and use them until they decide to re-engineer. This standard binding capability has made Ada much less objectionable to people with large legacy systems.

Fallacy: ***Ada tools are either too expensive or nonexistent.***

Fact: Ada tools are currently more expensive because the market is smaller. The AJPO has been actively working with industry to increase the number and availability of Ada tools, compilers, and environments by increasing the commercial Ada market. As the market gets bigger through the commercialization of Ada, there will be more venture capital for competitive tools and prices will come down. *[See the discussion on the **Ada Technology Insertion Program-Partnership** in Chapter 10, Software Tools.]* Note that for some platforms there are low-cost, or even free, high-quality Ada compilers. Also, be sure you are comparing equivalent capabilities; many users of other languages buy additional tools to check for errors that Ada compilers automatically detect at no additional charge.

Fallacy: ***It takes longer to program in Ada than it does in other languages.***

Fact: According to the AJPO, there are data to the contrary. The AJPO's philosophy is, the time allotted to program applications *independent of language* is too short. The albatross around most programmers' necks is impossible schedules. Neither C++ nor Ada is going to solve that problem. What will provide a solution is an incremental development paradigm — the method where we build a little, test a little, field a little, and repeat the process. *[See Chapter 3, System Life Cycles and Methodologies.]*

Fallacy: ***The reuse initiative is a waste of time. Ada code will not be reused. It will be put in a big pool or library and forgotten.***

Fact: The AJPO is not after code; they want to pool *architectures*. Placing code in a library is like putting a book in the Library of Congress. It just sits on the shelf and gathers dust. What the AJPO wants to do is bring the bestsellers to the user so when he goes to the library with his architecture he pulls the right books. As Reifer explains the concept is simple,

> *Bring the product-line manuscript concepts from industry into the Defense Department. There are places where reuse has shined like on the restructured **Navy Tactical Data System***

> *(NTDS) program. Look at the lessons-learned from the Army's* ***Field Artillery Tactical Data Systems*** *and the lessons the Swedish Navy has learned with its Corvette system. They made a major investment in the future when they built it in Ada. Now they can retarget the software built for one class of ships to other classes with* ***70% reuse.*** [REIFER95]

Fallacy: ***The majority of programmers prefer any language to Ada and for the most part are Ada illiterate and plan on staying that way.***

Fact: According to the AJPO, there are only a few pockets of Ada illiterates. Of course, it depends on whether you are talking about weapon systems or MIS. For example, the **Naval Command and Control Ocean Surveillance Center (NCCOSC) RDT&E Division (NRaD) Center** communications programmers, like many others, are ***firm*** believers in Ada. In the MIS area, Ada shines on the Army's **Sustaining Base Information Services (SBIS)** program, the first major user of Ada 9X *[now Ada 95].*

Fallacy: ***Ada is only good for building large weapons systems. Everything else is better done in a less complicated language.***

Fact: The services recognize the need for a world-class sustaining base of software, not differentiating between tactical and MIS software, which is counterproductive. Presently our sustaining base is antiquated and must be brought up to a level that meets the needs of the warriors in the field. We must migrate our *best* legacy systems up to world-class. With Ada's bindings *[discussed below]*, we can migrate legacy software to new applications — *without doing everything over from scratch.*

Fallacy: ***There really are not very many Ada programs either completed or in the works.***

Fact: The next section "*Ada: Who's Using It?*" answers this fallacy.

CHAPTER 5
Addendum A

How Long Does It Take to Learn Ada?

Eileen Steets Quann
President, Fastrak Training, Inc.

A COMMON QUERY

As a provider of Ada training for the last seven years, we are frequently asked the same question: ***"How long does it take to learn Ada?"*** Since this question continues to be asked, it suggests that the answer is neither obvious nor simple. I believe that the difficulty of providing a succinct answer springs from it being the wrong question. This becomes more apparent when the question is changed slightly.

Suppose I ask instead, *"How long does it take to learn algebra?"* Most people immediately see that the answer is, *"It depends."* What kind of a math background do the students have? What do they need to be able to do? What kind of mathematical aptitude do the students have? How confident are they of their ability to learn?

Concepts in algebra are taught in the middle schools and continue through advanced courses in college. When have they *"learned"* algebra? Depending on answers to the questions above, it could be argued that it takes between a few hours and many years.

The Students and Their Jobs

How do we answer the question about Ada? At Fastrak, we begin by asking the client two questions: *"Who are your students?"* *"What do they need to be able to do?"* If you tell me that they have a strong background in software engineering, a solid understanding of concepts like abstraction, information hiding, and encapsulation; that they are

comfortable with strong data typing, use Pascal now, are highly motivated to learn, and have good software processes already in place, I will give you one answer.

Suppose instead you say that they are high school graduates, with virtually no knowledge of software engineering concepts and little design experience; they are experienced with only one language and an old one at that (COBOL or FORTRAN). They are resistant to using a new language and will be working on a large system with poorly defined processes, addressing complex design issues. It doesn't take a rocket scientist to know that you'll get a different estimate. While these may be the extremes in the software industry, we encounter students at both ends of the spectrum and many in the middle.

What backgrounds and experiences make Ada easier to learn? I believe that two criteria exert a bigger influence on the individual programmer's ability to learn Ada than anything else. Because Ada is an evolutionary rather than a revolutionary language, we have discovered that the people who already know several languages, regardless of what languages they are, learn Ada most quickly. They draw upon a broader experience base, and they learn quickly by analogy.

Two Kinds of Maturity

A common perception is that younger people have an easier time learning Ada than the more *"mature"* audiences. My observation has been that people who are accustomed to learning new things continuously will learn more quickly. This is because they know how to learn, are not intimidated by something new, and have confidence in their ability to learn. The only *"old dogs who can't learn new tricks"* are those who think they are.

The maturity of their software organization is another important factor. We have found that the higher the maturity level of the software organization, the faster and easier they learn Ada and the more receptive they are to using it. They already understand and appreciate the importance of software engineering and process. More mature organizations also foster the learning environment, which addresses the other two criteria. (Perhaps if only Capability Maturity Model Level 3 organizations were allowed to use Ada, the answers would be much simpler.)

CHAPTER 5 Addendum A

Other Considerations

What do they need to be able to do? Do you need software engineers to design and develop systems in Ada or coders to implement the designs of others? Will there be someone to help them when they get stuck? Will they use sophisticated tools, or do they just need to compile and execute their programs?

What about the training media? Do you intend to use all classroom instruction, individual reading, computer-based training (CBT), on-the-job training (OJT), or some combination? Each has its strengths and drawbacks, its champions and detractors.

A RANGE, NOT AN ABSOLUTE FIGURE

The good news is that the answer to the Ada question has a more narrow range than the algebra question. The bad news is that, because of the state of the software industry, the limited training software developers have had in the past, and the large number of Level 1 organizations, the answer is often higher than people want to hear. This is not because Ada is so difficult to learn, but because they need to learn so much before they get around to learning Ada if our goal is to grow software engineers.

Here are a few guidelines to estimate training time. Some people will claim that these are high, others will argue that they are optimistically low. The estimates are for classroom training only; CBT would be shorter but generally more expensive because of the upfront development costs, follow-on OJT with knowledgeable mentors and good documentation is assumed but not included. The ranges represent the range of skill levels, motivation, and ability of typical students. Atypical students are outside these ranges. Table 5-1 shows typical course duration for teaching various skills.

SUBJECT	DAYS
Software engineering concepts and process	5 - 20
A new design methodology	4 - 10
Software development tools (per tool)	3 - 5
Basic Ada programming skills	5 - 10
Advanced Ada concepts for designing systems	5 - 15

Table 5-1 Software Engineering Course Length

The estimates are generally cumulative but may be reduced in a well-planned and well-executed training program. To define the needs of your organization, you must first answer the key questions: *"Who are the students?" "What do they need to be able to do?"* Your answers bound your question, ***"How long does it take to learn Ada?"*** *[NOTE: See Volume 2, Appendix A for information on how to contact Ms. Quann at Fastrack, Inc.]*

Ada: WHO'S USING IT?

We have all heard the question, *"Who's really using Ada?"* as well as its follow-on statement, *"No one uses Ada."* The truth is, *there are many major programs that chose Ada over other programming languages.* This section provides a collection of Ada examples from DoD (including all the service components) and industry (including major American and European programs) which should conclusively answer the question *"Who's really using it?"* Also included are discussions of other programs successfully using Ada:

- **Seawolf Submarine's AN/BSY-2 Project,**
- **NCTAMS-LANT Project,**
- **NCPII Re-Engineering Project,**
- **Theater Display Terminal (TDT),**
- **Ada for Windows Project,** and the
- **Intelsat I-VII Satellite Project** where an industry team chose Ada for a critical and complex industry program solely because of Ada's capabilities and the way it supports software engineering principles.

NOTE: If you are interested in more information on any of these examples, or would like to find additional examples, contact the Ada Information Clearinghouse *[see Volume 2, Appendix A for the address and phone number]* **or view information online** *[see Volume 2, Appendix B for the Web address].* **Now you will be prepared the next time someone says, "*No one uses Ada*!"**

Ada Use in DoD

A Survey of Computer Programming Languages Currently Used in the Department of Defense, published by the **Institute for Defense Analysis (IDA)**, also dispels the fallacy that Ada is not frequently used. The IDA report was based on 19914 data collected from 423 weapons systems programs and 53 major MIS programs, including R&D programs with budgets over $15 million and procurement programs worth more than $24 million. Collectively the sample systems amounted to 237.6 million SLOC. [HOOK95] [CONSTANCE95] Tables 5-2 (below) and 5-3 (below), taken from this report, indicate how Ada is gaining ground on older, better-established languages. Table 5-4 (below) provides a synopsis of Ada programs by service as published by the AdaIC.

3GL and VERSION	TOTAL SLOC REPORTED (in millions)
Ada 83	49.70
C 89	32.50
Fortran (pre 91/92)	18.55
CMS-2V	14.32
Jovial 73	12.68
C++	5.15
CMS-2M	4.23
Other 3GLs	3.38
Pascal (pre 0)	3.62
Jovial (pre J73)	1.12
Fortran 91/92	1.00
PL/I 76/87/93 subset	0.64
Pascal 90 (extended)	0.29
Basic 78 (minimal)	0.17
LISP	0.10
COBOL (pre 85)	0.09
COBOL 85	0.00
Total	148.38

Table 5-2 Total SLOC by General Purpose 3GL for Weapons Systems

3GL and VERSION	TOTAL SLOC REPORTED (in millions)
COBOL 85	14.05
COBOL (pre 85)	8.59
Ada 83	8.47
Basic 87/93	2.18
C++	2.05
C 89	1.55
Fortran 91/92	0.87
Fortran (pre 91/92)	0.47
Total	38.24

Table 5-3 Total SLOC by General Purpose 3GL for MIS

SERVICE	PROGRAM TITLE	DESCRIPTION	SPONSOR
ARMY	*FIREFINDER Block II Improvement Program*	Locates sources of hostile mortar, artillery, and rockets	Communications-Electronics Command (CECOM)
	Icarus	A discrete event simulation system intended for the study of organizational behavior	US Army Research Office
	Mobilization Readiness Priority (MRP)	Supports Army Reserve enlistment by matching training and units to the soldier's skill level	US Army Keystone Office
	HELLFIRE Tactical Missile (AGM 114)	The Army's primary anti-tank missile; air-to-ground, launched from helicopters, and laser-guided	US Army Missile Command
NAVY	*Ada Validation Tools*	These tools provide a general data capability for systems developed in an AdaSAGE environment	Naval Computer and Telecommunications Command
	F4-J Trainer	Simulates capabilities of an F4-J aircraft throughout its flight	Naval Training Systems Center
	Infrared Search and Track System	A passive IR systems within the F-14 Naval Air Systems aircraft designed for ID and track of airborne targets	Naval Air Systems Command
	Naval Intelligence Processing (NIPS)	Provides a worldwide characteristic and performance database for intelligence analysts	Space and Naval Warfare Systems Command
	Patrol Aircraft Test Laboratory (PAIL)	A generic test facility designed for the support of patrol aircraft and related missions	Naval Air Training Command
AIR FORCE	*Automated Weather Network Communications*	Provides communications software for AWN terminals, and enables the connection of Z-248s to AWN. AWNCOM consists of 6,500 lines of executable Ada code and was delivered in Apr 90	USAF Command and Control Systems Office
	Have a Glance	Development of a laser-based countermeasure system to prevent enemy missiles from intercepting friendly aircraft	Wright Laboratory

Table 5-4 New or On-Going Ada Programs Throughout DoD

SERVICE	PROGRAM TITLE	DESCRIPTION	SPONSOR
AIR FORCE	*Granite Sentry*	Modernizes hardware and software at Cheyenne Mountain for four functions: air defense, command post, battle staff, and weather	USAF Space Command
	Autonomous Guided Weapon	To control the weapon from aircraft release until impact on target so that the aircraft does not have to remain in the area until the target is hit	USAF Systems Command
MARINE CORPS	*Deployed Check Issue Processing*	Microcomputer application for disbursing officers to produce paychecks at field sites	Headquarters, Marine Corps
	Woman Marine Model Prototype (WMMP)	IBM AT/386 bases model to determine optimal distribution for female marines based on USMC set policies. WMM is capable of simulating *war-time* situations	USMC MEMPAR

Table 5-4 New or On-Going Ada Programs Throughout DoD (cont.)

Large-Scale Commercial Applications from Around the World

Table 5-5 lists large-scale commercial applications where Ada is the language of choice. Note the use of Ada on Boeing flight control software, an extremely safety-critical application.

Seawolf Submarine's AN/BSY-2 Project

The **Submarine Combat System for the USS Seawolf** (SSN 21), the **AN/BSY-2**, is the largest mission critical, real-time shipboard software acquisition in the history of the US Navy. According to **Rear Admiral Scott L. Sears**, commander of the Naval Underseas Warfare Center, not only is the BSY-2 program the largest Ada development in DoD, it is one of the largest in the world — and it is ***"truly an Ada Success Story!"*** The 6 million plus line-of-code acquisition (3.5 million of which is tactical) is turning the corner on DoD's legacy of software-intensive fiascoes because the Navy pioneered the use of Ada and because *"Ada provides the best intellectual control for the development of large systems."* [SEARS95]

Ada SYSTEM DESCRIPTION	DEVELOPER/SPONSOR
Integrated CAD engineering system links CAD tools that can manipulate information in an object-oriented database using a uniform user interface	Byron Informatik, Germany and Switzerland
Financial management system includes an integrated relational database, a real-time ticker data display, and vectored optimization calculations	Wells-Fargo Nikko Investment Advisors in San Francisco, California
Process control software in a $200-million steel rolling mill	General Electric Corporation
Geophysical data processing system for analyzing information collected during world-wide explorations	Shell Oil Corporation
Terrain and **position location systems** that give upcoming topographical information to train engineers	Rockwell International
On-line banking systems	Nokia Information Systems, Finland
Telecommunications control systems	Dutch Post and Telecom
Multi-state payroll systems	MAN Truck/Bus
Nuclear plant control systems	SEMA (France)
Oil exploration simulation systems	Schlumberger
Automated warehouse and **industrial systems**	BT Systems, Moindal, Sweden
Embedded systems for satellite, international space station robot arm, and air traffic control	Computer Resources International, A/S Space Division (CRI), Birkerod, Denmark
Seekers, engineering tools, electronic warfare, space systems, and **flight controls**	Dassault Electronique, Saint-Cloud, France
Purchasing/maintenance systems for equipment and supplies for the Swedish air force, army, and navy	Forsvaret Materielverk, Swedish Defense Material Administration (DMV)
Communication, information, and **scientific systems**	Heise Software, Maintal, Germany
Automated warehouse and **industrial systems**	SATT Control, Maimo, Sweden
Embedded systems for helicopters, aircraft control, and traffic control	Sextant Avionique, Villacoublay-Cedex, France
Surveillance, guidance, and **communication systems**	TERMA Elektronik AS, Surveillance Systems Division
Electronic steering/braking systems for aircraft and high-speed trains	Thompson, CSF, Brake and Steering Division, Paris, France
Automobile manufacturing systems	Volvo, Götenborg, Sweden
Electronic mobile exchange (EMX) cellular switching system connects mobile units with numerous base site controllers and base stations	Motorola's, Radio-Telephone Systems Group, Cellular Infrastructure Division
Financial transaction system provides world-wide access to the Chicago Mercantile (Merc) Exchange and other major futures and options trading centers	Reuters International
Railroad signalling and **train control system**, Astree, provides location, speed, distance, switch, operations, and safety information controlling almost 100 trains over 600 kilometers of rail for France's national railroad association	Societé Nationale des Chemins de fer Français

Table 5-5 Large-Scale Commercial Ada Systems

Ada SYSTEM DESCRIPTION	DEVELOPER/SPONSOR
Electricity monitoring system integrates information received from many switching stations increased processing time over old Fortran system by 99.5%	Bernische Kraftwerke, Berne, Switzerland
Medical diagnostic system analyzes up to 750 absorbency and fluorescence photometry and ion selective electrode tests through order and result processing and calculation, data storage and retrieval, real-time instrument control, test scheduling, event tracking, server interfacing to laboratory database, quality control, system configuration, maintenance, and diagnostics	Tegimenta, subsidiary of Hoffman-La-Roche AG
Automated banking system that processes transactions on interest-bearing accounts between 350 workstations and 1.2 to 1.4 million customer records/ accounts, handling 10,000 modifications, 20,000 inquiries, and 7,000 check orders per day	Kukobeza, Switzerland
Manufacturing process supervisor, OSCAR II, enables the control and management of every type manufacturing process through real-time display of machine status data and overall process coordination	Delta Technologies, France
Radio telescope control for 15-meter microwave telescope through computing azimuth and elevation commands for the antenna position server for the Metsahovi Radio Research Station, Kylmala, Finland	Helsinki University of Technology
Mobile communication system performs access control, monitoring and alarm handling antenna steering, man-machine interface, and protocol handling for mobile satellite system terminals	ABB Nera, Sweden
Integrated circuits industrial design system compiles high-level parametrized schematics for a given macrocell into low-level placement orders of microcells and routing orders of interconnections	Dolphin Integration
Automatic train control system, TVM 430, displays instructions for train engineers and checks that they are properly executed for the French National Railroad linking Paris and Lille and for the future Transchannel link	CSEE (Compagnie des Signaux)
Automobile assembly robot qualifying simulator models real-time operations of any production machine and provides a description and analysis of manufacturing process control systems behaviors simultaneously for French car maker, Citroen	3ip (small auditing and consulting company)

Table 5-5 Large-Scale Commercial Ada Systems (cont.)

Ada SYSTEM DESCRIPTION	DEVELOPER/SPONSOR
Vehicle engine test system, CHASE, acquires, processes, displays, and archives high-speed data from engines under test conditions	Ricardo Consulting Engineers
Power management system governs sequence of events in the power ignition process regardless of external circumstances such as weather or unforeseen deficiencies for military aircraft and the Boeing 777	Sunstrand, Rockford, Illinois
Flight control software used in the 747, 767, and 777 airliners	Boeing Aircraft Company
Cockpit displays for the 747-400 airliner	Boeing Aircraft Company
Flight warning system monitors aircraft systems, detects failures and dangerous flight conditions, and generates corresponding warnings	Aerospatiale, Avionics and Systems Division
Video tape editor copies tapes to disk, edits them with random access speed, and copies results back to tape	Decision Aids, Saratoga, California
Virtual reality training simulations, Paintball and Fireball, integrate real-time interaction and visual simulation for military and commercial task training applications	Silicon Graphics, Inc.
Air traffic safety system simulator models different air traffic patterns enabling researchers to study new sections of air space and new monitoring procedures	Eurocontrol (independent European organization overseeing air navigation safety)
Automatic metro piloting system, METEOR, controls train traffic, regulates train speed, manages alarm devices, and allows operation of automatic and non-automatic trains on the same line for the Paris Transportation Authority	GECAlsthom (designed train system) Matra Transport (designed automated systems)

Table 5-5 Large-Scale Commercial Ada Systems (cont.)

The BSY-2 is an advanced, highly complex Ada system designed to handle all the signal and data processing of the Seawolf's inboard electronics suites. It will take real-time data from thousands of sensors and convert, sort, and analyze those data to produce meaningful decision-coordination information to be used by 20 officers and technicians in the attack center by synthesizing ship maneuver and weapons deployment actions. The goal of the BSY-2 is to: (1) enable the submarine to detect and locate targets quicker, (2) allow operators to perform multiple tasks and address multiple targets concurrently, and (3) ultimately reduce the time between detecting a threat and launching weapons. [GAO89]

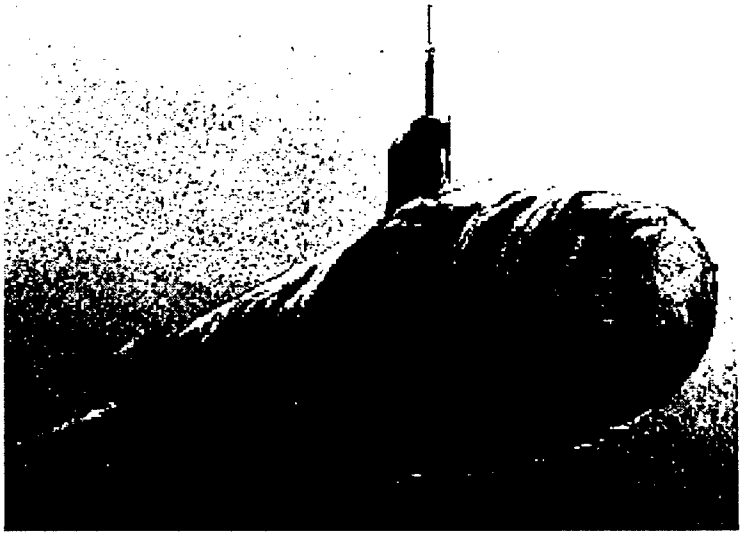


Figure 5-3 Software-Intensive SSN-21 Seawolf Attack Submarine

Demanding Software Requirements

The software requirements are demanding in that the system must be highly flexible and resilient in its ability to deal with unprecedented scenarios and data of unknown accuracy subject to erratic acoustic transmission through water. Software reliability and fault tolerance demands are also extremely high considering the cost and ramifications of a software failure while, for example, tracking an incoming torpedo. As RADM Sears explains, *"If you have a software failure such that the system crashes, then the entire time you are bringing the system back up you are essentially running blind, deaf, and impotent!"* [SEARS95] Also, because attack submarines are at sea months at a time with the combat system operational 24-hours a day, if the system fails it must be correctable at sea, or the ship's mission must be aborted to return to port. Therefore, the system must provide for online diagnosis and fault location, graceful degradation and reconfiguration in the event of casualty, and an on-board training capability that provides realistic scenarios.

Stealth submarines only number in the tens, therefore, the software-intensive systems we build for them are unique, costly, and with expected useful lives of around 20 to 30 years. During that life time, hardware technology will turn over perhaps ten times, mission requirements will evolve, threats will change, and weapons systems

will advance. Thus, the **software architecture** has to permit growth and change in an incremental, modular way. It must also provide intellectual control over an enormous, technically-challenging system and support concurrent development by multiple and diverse teams.

World's Biggest, Unprecedented Ada Development

The BSY-2 development contract was awarded in FY88, during the height of the Cold War when Ada 83 was still in its infancy. There were virtually no validated Ada compilers, especially none that would handle a real-time system with millions of lines-of-code. There were no proven CASE tools, no Ada-experienced software engineers, and no reuse libraries. No one had ever attempted to develop a software system of this magnitude and complexity. It included a new computer, a new operating system, a new programming language, a new user interface, a new distributed architecture, a new database approach, and a new design methodology — all to be developed in accordance with a new military standard, DoD-STD-2167. The contractor's Software Development Plan (SDP) called for up to 900 software personnel, spread over seven years, with seven development organizations under the direction of the prime contractor. They proposed to partition the system into 113 work packages (or building blocks) each containing up to 75,000 lines-of-code. They also proposed an aggressive training program for both Navy and contractor personnel on software engineering principles and use of Ada.

Challenging COTS Integration

To save on development costs, the Navy wanted to incorporate as many **COTS products** into the acquisition as possible. As Ada became more popular in the commercial market, CASE tools began to appear, but there were still few from which to choose. The main issue they faced with COTS products was scalability. Most of the tools out there were designed to work with around 50,000 source lines-of-code (SLOC). With each work package over 75,000 SLOC, they broke about every tool they attempted to use. Another issue was the real-time processing requirements. Few tools were designed to handle the durability and reliability criteria of a combat system. The contractor was finally able to piece together an interoperable set of commercial tools using software they built in-house linking them together.

The COTS database manager for the tactical system they chose was the fastest on the market. Even after extensive work with the vendor, they could never get the performance needed and the contractor had to build their own database manager out of Ada. In this instance, it was quicker and cheaper to build in Ada than to try to modify a COTS package. They had better luck with the commercial run-time environment supplied by the compiler vendor where the only modifications needed were for BSY-2 and Navy-specific hardware. The vendor was successful in working closely with the BSY-2 team in fixing problems and incorporating needed changes to their product. Due to this close relationship, the BSY-2 system tracked in lockstep with the vendor's commercial versions until the Navy chose to baseline their system and freeze versions.

According the RADM Sears, the main thing they learned was that COTS software and hardware are not free. As these experiences with COTS illustrate, savings are only possible if the COTS product embodies requirements that closely reflect those of your application, especially where real-time performance is critical and exacting. In some cases it may be cheaper to modify a COTS package rather than start from scratch, but this is a make-or-buy decision that must be made carefully. [SEARS95]

NOTE: See Chapter 10, *Software Tools*, for an in-depth discussion on the BSY-2 tools integration lessons-learned. Also see Chapter 14, *Managing Software Development*, for a discussion on why modifying COTS is NOT usually advised.

Ada and Engineering Discipline Are Key to Success

The basis for this Ada success story is ***software engineering discipline***. They proved that the larger the system and the more demanding its performance and reliability requirements, the more essential engineering discipline is to delivering a quality product. Ada provides the best intellectual control available today for managing the development of huge software systems through its packaging concept, strong typing, and separate compilation support.

CHAPTER 5 Ada: The Enabling Technology

BSY-2 Ada Lessons-Learned

The Ada-specific lessons-learned on the BSY-2 program include the following.

- Ensure memory utilization estimates include *stack* and *heap* along with the expanded source.
- Fully exercise target code units to ensure they perform as intended on the host environment.
- Debugging processes and compilation dependencies will be significantly reduced if nested subprogramming is avoided.
- Group functionally-related messages into a single package to reduce start-up overhead and lessen dependencies.
- Ensure all loops include a "*time-out*" condition for preventing resource deadlocks.
- Enhance time-critical processes through the application of **Rate Monotonic Scheduling** techniques.
- Allow adequate ramp-up time for familiarization with the CASE tool environment.
- Choose a mature compiler. Develop compiler performance measures (CPM). Establish a strong relationship with the compiler vendor. Schedule compiler upgrades.
- Avoid Ada-incompatible *single-thread-of-control* by isolating COTS packages from Ada code through tasking or separate operating system processes.
- Include source code rights in COTS licensing agreements. ***Avoid modifying COTS!***
- Establish regression test procedures for COTS version change.
- Investigate COTS defect handling mechanisms and establish a mapping to the application program defect model.
- Establish Ada exception handling methodology early and a global defect model for a more robust system.
- Ensure explicit type checking is performed on external parameters.
- Maintain control of data through Ada information hiding techniques.
- Prototype message transfer schemes early in design. Use partial tasking where feasible to reduce Ada rendezvous overhead times.
- Establish policies for dynamic memory management.
- Factor actual and prototype data into timing analyses.
- Prototype the target environment.
- Ada experts should take the lead in solving complex problems and in establishing usage guidelines/procedures.

NCTAMS-LANT Project

An example of a successfully completed Ada program is the **Navy Computer and Telecommunications Area Master Station — Atlantic (NCTAMS LANT)** system. Their learning curve was painful, but in 1994, the program received the award for the *Best Object Based Application Developed Using Non-Object Tools* at the Object World Conference international competition. It took three years to re-engineer the Type Commanders Headquarters Automated Information System (THAIS) from COBOL to Ada using the AdaSAGE development environment. *[AdaSAGE is discussed in Chapter 10, Software Tools.]* With the re-engineering came the new name, **TYCOM Readiness Management System (TRMS)**. The original THAIS consisted of 2,000 programs and 2.8 million lines of COBOL. It was a big program and the award marked a major accomplishment.

NCPII Re-engineering Project

The **Fleet Interface for Navy Communications Processing and Routing** is another program currently undergoing re-engineering to Ada. The re-engineered system, called NCPII, has been designed using object-oriented development (OOD). *[See Chapter 14, Managing Software Development, for a discussion on OOD.]* NCPII integrates a variety of COTS products: a trusted UNIX B1-Level secure operating system; a trusted relational data base management system (RDBMS); a trusted X-Windows system using compartmented mode workstations and software; and a trusted local area network. NCPII will take advantage of NIPRNET to connect to worldwide operational sites. *[See Chapter 14, Managing Software Development, for a discussion on trusted and secure systems.]*

Theater Display Terminal

The story of the Air Force's **Theater Display Terminal (TDT)** during the **Gulf War** is another example of why using Ada is smart. The TDT is a rapidly deployable missile warning system written in Ada for a Sun UNIX workstation environment. During the conflict, the US deployed the TDT to theater to warn of imminent Iraqi SCUD attacks. On January 11, 1991, CENTCOM personnel realized they needed to know the *country-of-origin* for all incoming SCUD missiles. Fighting in a volatile part of the world with varying and diverse levels of threats, the task force commander required quick

determination of whether a SCUD was launched from Iraq or from some other hostile country. From January 12th to the 13th, Air Force Space Command revised an existing Ada *country-of-origin* algorithm, along with an Ada geopolitical database. On January 13th, these assets were integrated into a developmental system and on January 14th the enhancement was integrated into an operational system. On January 15th, the software modification was flown to theater and installed for use on the TDT. On January 16th, ***just 5 days after the requirement was identified***, the new capability was up and running and was successfully used during an Iraqi SCUD attack on Israel!

The capability to expeditiously support rapidly changing, real-time threat scenarios was only possible because the TDT, the *country-of-origin* algorithm, and the geopolitical database were written in Ada. Integration of these assets was possible because of the **Ada package**, which provides clean logical interfaces between modules. Had Ada not been used, the cost to support and develop code to fulfill the new requirement would have been substantial. The enhancement surely would not have been fielded within 5 days! The TDT experience proves that using Ada to produce high quality, supportable software for large, safety-critical applications is smart business.

Ada for Windows Project

In May 1993, **First Lieutenant Shannon Straffin** was one of five programmers from the **Advanced Computer Systems Flight** of the Air Combat Command (ACC) Computer Systems Squadron (formerly the 1912th Computer Systems Group) who was given the task to ***develop Microsoft Windows programs quickly in Ada!*** The difficulty of the task is clearer when you realize that nearly all Windows development, and therefore, nearly all Windows training and support tools, are written in C. Ada for Windows was an unheard of concept that was largely considered not feasible.

Team Training Was An Innovative Effort

The programmers had been successful in developing DOS-based programs, several of which are being used throughout ACC today. However, their combined Windows experience was one person receiving 10 days of training at Microsoft University. After some hesitation, the newly developed Ada Windows team tackled the assignment enthusiastically. Using the products they had in hand — Meridian's Open Ada for Windows compiler, Borland's Resource

Workshop and Microsoft's Software Development Kit — the team began a *proof of concept* effort.

The first hurdle was the transition from DOS to Windows. *"Windows programming is very different from DOS programming,"* Staff Sergeant Jeff Whippo, who attended the Windows training, said. *"You have to unlearn a lot of what you already know,"* he said. *"It takes a good two to three months of constant effort to really get the hang of the new environment. We had to acquire a talent to 'read in C and write in Ada,' because all the Windows support is geared toward development in C,"* he said. This second hurdle, lack of applicable training, was overcome with time and teamwork. Armed only with their product documentation and Charles Petzold's, Programming Windows 3.1, team members were ready to begin applying their new skills to the job at hand.

Lack of Ada Windows Tools Sparked More Innovation

The team chose to rewrite a DOS program it had just developed, the Sarah Message Distribution System, for its training program. Since the program's logic was already written, the team was free to concentrate on the Windows implementation details. In the process, they discovered a third hurdle — a complete lack of Windows development tools for Ada. *"There were no Ada tools available to do what we needed to do, so we built some of our own using existing Windows products written in C,"* **Master Sergeant Dave Jaffe**, Non Commissioned Officer in charge of the team, said. *"We were forced to rely on our own ingenuity,"* he said. *"Once we figured out that we could rewrite C header files and data structures in Ada to make C functions available to our programs, things became much easier."*

Team members made this discovery when they decided to improve on SMDS's initial design by writing a Ada interface from their program to Microsoft's Messaging Application Programming Interface (MAPI). This would enable the program to automatically mail messages directly to recipients. The effort paid off. After only four months, the team produced an undeniable success! Not only had team members proven they could write Windows programs in Ada, but they had improved on the design of their own product, saving their customers an additional two hours of work per day.

Lessons-Learned Applied to Other Efforts

The lessons-learned from MAPI worked well when applied to other efforts. The team wrote an interface for functions in the Borland C Run-time Library. This interface has not been completed, but the functions used most often are included. Adding access to another function, if needed, is a matter of adding a two-line entry into the existing interface. In addition, Senior Airman (SrA.) Jerry Wimer and Airman First Class Ken Foreman have developed Ada interfaces for Sequiter's Codebase database engine for both FoxPro and Clipper style databases. This allows their team to develop databases without having to hand code common database functions, such as record locking and indexing.

New Skills Breed New Applications and Training Programs

The Ada Windows team has since developed an automated phone directory, an electronic bulletin board, and a time tracking system. The team has also created dynamic link libraries, some of which have been used in FoxPro programs. Senior Airman. Curtis Worthington has documented the lessons his team learned and has developed training materials. He trained five other programmers in the new technology and is training two other programmers new to the flight.

Ada Windows Team Spreads the Wealth and Presses On

Word of the Ada Windows team's accomplishments is spreading. Five people from **NCTAMS LANT** program *[discussed above]* in Norfolk, Virginia, visited in June. *"We're looking to move into Windows programming in Ada and we heard abut the work the Air Force had done through a vendor who was trying to sell us both the same product,"* David Cuneo, head of the NCTAMS LANT software engineering branch, said. *"It was great to get in touch with them and see what they had come up with,"* he said. *"There are so few people even attempting this sort of thing, which makes it so important for us all to be in contact with each other."* During the visit, Cuneo's group saw a demonstration of the time tracking system developed in Ada for Windows and discussed the lessons-learned during its development. The Navy people took a copy of the reuse library and the training materials developed by the Ada Windows team with them.

The direction of the Ada Windows team will remain the same: ***master the best of new technology and use sound software engineering practices to provide the customer with the best possible automation solution.***. Wimer has begun the work need to come up to speed in the new environment by writing the interface to the Microsoft SQL server and developed a test Ada program which can communicate with the server.

Despite this initial success, there is still much work to be done before the team will master the necessary skills. Working in ACC's premier software development flight is demanding but rewarding. The Ada Windows team has done what many people said could not be done. Though the team has come a long way in the past year, team members will be the first to admit that there is still much for them to learn. *[See Volume 2, Appendix A for information on how to contact Captain Randy Powell.]*

Intelsat I-VII Satellite Project

Intelsat is an international telecommunications consortium of approximately 122 members that builds and manages communication satellites. Since its first satellite was launched 27 years ago, it now has 18 operational satellites in orbit. In 1993, it launched its new-generation satellite, Intelsat I-VII — the first satellite developed completely in Ada. This extremely large software development is another example of why Ada was chosen because of its support for software engineering principles. ***Ada was also chosen because it promised the necessary robustness, flexibility, reliability, and maintainability required by a demanding cosmic-radiated, space-based environment.*** [RIEHLE94]

When Loral Space Systems won its contract to develop the I-VII in 1990, they selected the MIL-STD-1750A computer architecture because it had a standardized software instruction set across all implementations and was radiation-hardened. Although basically autonomous, ground stations need access to I-VII onboard software to upload modifications and upgrades throughout its life cycle. Thus, one of the software requirements was a command function which moves antennae, disables certain operational features, fires thrusters, and uploads new memory values. Other functions include operating system software (scheduler, power-up, and fault detection), attitude control and determination (one of the most difficult algorithmic functions), battery and thermal monitoring and control, telemetry transmission, and overall control safety.

Ada's Package Feature Benefits

According to Richard Riehle, from AdaWorks (a firm hired to teach Ada software engineering methods to the development team), Ada's package feature greatly contributed to the success of the program. Ada programmers, not restricted to working solely on the VAX, RS/6000, or Sun workstations, were able to do a considerable amount of creative programming. Because Ada is the same language no matter which processor used, programmers were able to write and compile Ada code on smaller, less-expensive MS-DOS PCs which was later merged into larger applications. Allowed to independently and freely develop code, programmers were able to experiment and test new ideas to achieve optimum results. Package components were divided into separate compilation subunits which were worked by individual team members. These independent units (packages and subunits) contributed to reliability. In addition, peer inspections *[discussed in Chapter 15, Managing Process Improvement]* are easier to conduct on smaller chunks of code.

The Ada method of task decomposition differs from that found in C, C++, or Pascal. The Ada compiler environment keeps track of the various components and enforces discipline on the process of creating, compiling, and linking application components. This is why Ada is best for large-scale programs, such as the I-VII, that require large design and programming teams, have a high degree of complexity, must be reliable, and are characterized by a long life cycle. The Ada compilation environment also provides a considerable amount of configuration control. [RIEHLE94]

Ada's Typing Feature Benefits

Ada compilers check the consistency of every data name in a system to reduce the likelihood of incorrect assignments. They also perform syntax checking which makes programmers follow a stringent set of rules rigorously enforced by the compiler. The Ada compiler prevents collisions between variables having incompatible types, alerts programmers of potential out-of-range conditions, and ensures consistency among actual and formal parameters in subprogram calls. As Riehle explained, Ada's typing feature prevented square pegs from going into round holes, and even prevented round pegs of the wrong size from going into round holes — an important safeguard for a highly reliable system. [RIEHLE94]

Why Ada for Intelsat I-VII?

When a application is large, complex, must perform under rugged conditions, with a minimum number of defects, Loral realized knowledgeable software engineers choose Ada. Although C is a simple, easy-to-understand language, it was not chosen because it allows programmers to write code in any style. It contains poor intrinsic support for consistency checking across modules and permits function calls with incompatible data types or incomplete parameter lists. It is heavily dependent on the use of pointer types, but allows pointers to behave in undisciplined and bizarre ways. Maintenance was also a major consideration in Loral's choice of Ada. Satellites must remain operational for sometimes decades, but as technology and requirements change, their code must be modifiable. The more years that go by since the application was originally written, the harder it is to maintain. Ada puts demands on designers and programmers to create code that is easy to maintain. The big difference between Ada and C is that Ada takes the long-term into consideration, whereas C looks at developing code as a *quick and easy* solution.

Ada proved to be an investment in the future. The next Intelsat effort, Intelsat VIII, will reaffirm the consortium's commitment to Ada. Loral is so proud of the Ada design and code it used on I-VII, it is leveraging that success to build another satellite, NSTAR, for Nippon Telephone and Telegraph in Japan. *The new satellite will reuse much of that same Ada code.* In addition, Intelsat has commissioned Loral to build a follow-on satellite, I-VIIA, also to be programmed in Ada. Ada in space systems is flying high! [RIEHLE94]

ACHIEVABLE SUCCESS WITH Ada

The benefits of using Ada all relate to the factors that can make or break a software development program. Chapter 1, *Software Acquisition Overview*, emphasized that in the past performance, cost, and schedule failures on major software-intensive programs were usually traceable to the **software component**. What if you could find a way to increase programmer productivity, reduce defects, increase performance efficiency, keep within projected life cycle estimates, and reuse existing code? You could forge ahead in new directions by reversing demoralizing past trends. Ada, in concert with sound software engineering practices, could well be your opportunity to accomplish these goals. Data collected on 75 industry Ada

programs, representing 30 million lines-of-code that indicate these goals are achievable, as illustrated in Table 5-6. Considering the substantial sample size, these figures are quite impressive.

GOAL	INDUSTRY RESULTS WITH Ada
Productivity	• 10-20% degredation during first program 20% improvement attainable on subsequent programs vs. other languages • 17% improvement attainable on Ada programs through environment/tool integrated with object-oriented methods
Quality	• 20-30% fewer defects than other languages on first programs • Maintainability significantly improved with 10-20% lower cost
Performance	• Ada programs 10% smaller than other languages using conventional packaging methods • Bench marked Ada programs ran as quickly as others in 70% of the cases
Development Time	• Ada had no effect on development time • Distribution changed from 40% design, 20% code, 40% test (traditional) to 50% design, 10% code, 35% test (with Ada) • Incremental or spiral models reduce development time 20-30%
Reuse	• 18% average today compared with 10% 3 years ago (10% needed to break even given cost of reuse)

Table 5-6 Industry Experience with Ada [REIFFER92]

Ada versus the C++ Challenge

You might ask, "*Is Ada critical to obtaining the advantages found in the engineering of software? Are there not other languages that might be just as good as, or better than, Ada?*" In the spring of 1991, the Air Force accepted this challenge and conducted a study, *Ada and C++ Business Case Analysis*, [Ada/C++91] to answer exactly these questions. Before going very far, it was determined there are only two serious languages for developing engineered software today: Ada and C++. A group of recognized experts from the software development community (DoD and industry) was formed to determine under what circumstances the DoD policy of exclusively using Ada should be waived in favor of C++. They looked at the issue from various perspectives: quality and quantity of tools and educational support, technical language factors, actual quantitative experience reported, and cost-effectiveness modeling a commercial enterprise might use. ***The results, even when comparing the 1983 version of Ada to the current C++, were decisively in favor of Ada!*** *[You can obtain a copy of this report through DTIC or from the STSC.]*

Tools, environments, and training. When comparing Ada tools, environments, and training with those of C++, these support elements were considerably more mature for Ada. In 1991 there were 28 US companies supplying **validated Ada compilers**, whereas only 18 vendors offer C++ compilers. Ada compiler validation is a rigorous process, and is possible because Ada is standardized and has a validation suite that compilers must pass. *[NOTE: On 15 Feb 1995, Ada became the first internationally standardized object-oriented programming language, (ISO/IEC 8652:1995).]* In contrast, there is an effort to make a standard version of C++; however, that has not yet happened. Since it is not standardized, there is no validation for C++ compilers.

Both languages are supported on PCs and workstations. Regarding software engineering tools, a variety were found for both languages. Ada is taught in 43 states at 223 universities and 13 DoD installations. In 1991, C++ was taught in four states at four universities and at no DoD installations. Use of C++ has grown since 1991 and a wide range of training is now available. However, unlike Ada, which is an ANSI, an ISO, and a FIPS standard, C++ is only a draft standard. In 1985, Ada was evaluated by the **FAA** by comparing 48 **technical language features** (criteria) in six categories with four other languages. They concluded the long term benefits of using Ada were significant, as illustrated in Table 5-7.

CATEGORY	MAXIMUM SCORE	Ada	C	Pascal	JOVIAL	Fortran
Capability	16.7	16.1	9.6	10.4	7.6	3.9
Efficiency	16.4	8.0	11.8	10.8	11.0	11.1
Availability/ Reliability	22.6	21.5	11.6	14.5	15.6	10.3
Maintainability/ Extensibility	17.4	14.0	10.2	12.2	6.8	8.3
Life Cycle Cost	11.3	8.2	7.4	7.8	4.9	5.2
Risk	15.6	8.8	8.9	7.6	9.6	8.2
TOTAL	100.00	76.6	59.6	63.3	55.5	47.0

Table 5-7 FAA Weighted Scores for 6 Criteria Categories

In 1991, a follow-on study was conducted by the **SEI** using the same methodology as that used for each of the 48 criteria as the FAA study for MIS and C3 applications. The 1991 weighted scores for the six categories are shown in Table 5-8.

CATEGORY	MAXIMUM SCORE	Ada	C++
Capability	16.7	15.3	11.3
Efficiency	16.4	10.7	10.9
Availability/ Reliability	22.6	19.1	12.6
Maintainability/ Extensibility	17.4	13.6	11.4
Life Cycle Cost	11.3	8.4	8.0
Risk	15.6	11.7	9.8
TOTAL	100.00	78.8	63.9

Table 5-8 SEI Weighted Scores for 6 Criteria Categories (MIS/C3)

The conclusion reached by the team of experts conducting the language features study was that today (as was the case in 1985 with C) Ada is significantly more capable than C++. The relative efficiency of Ada has improved. Ada still scores higher in availability/reliability. The Ada advantage in maintainability/extensibility persists; and Ada has attained a significant advantage over C++ in lowered development risk.

Productivity and cost. A comparison was made on available productivity and cost data for Ada and C++. Four categories of applications were analyzed: environment/tools, telecommunications, test (with simulators), and other. All programs analyzed were new developments with an average size of about 100,000 source lines-of-code (SLOC). The results of the productivity analysis are illustrated on Table 5-9 (below). The average cost across the four program categories are listed on Table 5-10 (below). *[These study figures tended to be skewed in favor of C++ because the team did not find C++ programs of equal difficulty with the Ada programs analyzed.*

	PRODUCTIVITY (SLOC/mm)	NUMBER OF DATA POINTS
Norm (all languages)	183	543
Average (Ada)	210	153
Average (C++)	187	23
First project (Ada)	152	38
First project (C++)	161	7

Table 5-9 Productivity Study Comparison (source lines-of-code/manmonth)

	COST ($/SLOC)	NUMBER OF DATA POINTS
Norm (all languages)	70	543
Average (Ada)	65	153
Average (C++)	55	23

Table 5-10 Cost Study Comparison (dollars/source lines-of-code)

be in accordance with military standards and incorporate formal reviews, additional documentation, and engineering support activities, whereas the C++ programs did not.]

This study also included an analysis of **integration defect rates** [defects detected from start of integration testing through completion of software **Formal Qualification Testing (FQT)**] and FQT defect rates (only those found during the FQT process). The results of the defect rate analysis are illustrated on Table 5-11.

The conclusion the team of experts reached from the productivity/cost study was that the ***standardization maturity of Ada is important.*** While Ada has a firm, well-policed standard, a stable C++ language specification has not yet been established. Individual vendors of the 18 C++ compilers and associated CASE tools will continue to offer their own enhanced versions of their products, making portability and reuse of C++ among commercial products difficult in the near future.

	INTEGRATION DEFECT RATES (defects/KSLOC)	FQT DEFECT RATES (defects/KSLOC)	NUMBER OF DATA POINTS
Norm (all languages)	33	3	543
Average (Ada)	24	1	153
Average (C++)	31	3	23

Table 11 Integration of FQT Defect Rates

Additionally, Ada programs have reported 15% higher productivity, with increased quality for double the average program size, compared to C++. If these data are normalized to comparably sized programs, the ***Ada productivity advantage is about 35%***. It was concluded that C++ still needs significant maturation before it is a low-risk solution for any large DoD applications.

Corporate cost analysis. The corporate cost analysis performed by the team of experts was based on a typical real-world MIS/C3 systems program. They defined a set of maximally independent criteria, judged each language against those criteria, and translated those judgments into cost impacts to emphasize the importance of each criterion from a life cycle cost perspective. The rankings of the two languages, based on this analysis, are illustrated on Table 5-12 (below). *[0 = no support, 5 = excellent support.]* The total scores represent a weighted sum of the rankings based on weights determined by the expert panel. The conclusions reached by this study indicate that the typical advantage of using Ada over C++ is life cycle cost savings of ***35% during the Development Phase and 70% during the Maintenance Phase***. It is expected that the new Ada 95 will tip the scales even further in Ada's direction. [GROSS92]

Historical fault rates. Another study, conducted independently of the *Ada versus C++ Business Case*, is worth mention in this context. This study compared Ada historical fault rates with those of C (the predecessor to C++). Ada was found to have a fault rate of 0.5, compared to that of 4.0 with C. This is an 8 to 1 difference. For example, where C would result in 8 faults per 2,000 lines-of-code, Ada would have only 1. [SBM93]

	Ada	C++
Reliable s/w engineering	4.5	3.2
Maintainable s/w engineering	4.4	3.2
Reusable s/w engineering	4.1	3.8
Real-time s/w engineering	4.1	2.8
Portable s/w engineering	3.6	2.9
Runtime performance	3.0	3.6
Compile-time performance	2.3	3.1
Multilingual support	3.1	2.4
OOD/abstraction support	3.9	4.6
Program support environment	4.1	2.1
Readability	4.4	2.9
Writeability	3.4	3.5
Large-scale s/w engineering	4.9	3.3
COTS s/w integration	2.8	3.6
Precedent experience	3.6	1.5
Popularity	2.8	4.0
Existing skill base	3.0	1.8
Acceptance	2.5	3.3
TOTAL SCORE FOR MIS (Ada score is 23% higher)	**1631**	1324
TOTAL SCORE FOR C3 (Ada score is 24% higher)	**1738**	1401

Table 5-12 Corporate Cost Effectiveness Analysis

Ada versus the Assembly Challenge

In 1991, using another company's advice, a DoD contractor *[name withheld]* assumed assembly language *[described in Chapter 2]* would be the best choice for a small communications application [about 2,000 lines-of-code (LOC) for a TI 320C15 digital signal processor (DSP) chip]. Although Ada was specified in the contract, the contractor convinced the DoD program manager that compiled Ada code would be slower and would consume 3.5 times as much ROM as assembly code would. The contractor, permitted to use assembly, completed the software application in about 18 months. The first version did not run fast enough so it had to be tweaked to attain the required performance.

The DoD focal point and weapon systems software integrator, upon reviewing the contract, noted that Ada was not being used as stipulated. He was curious if Ada code could be used for this application and if it could meet the systems requirements of occupying no more than 2K of ROM within a 500-microsecond run-time. He spoke with numerous experts: a chip manufacturer, an Ada vendor, an independent consultant, a government laboratory, and the **Air Force Institute of Technology (AFIT)**. The experts concurred that Ada could not be categorically ruled out for this application. He informed the DoD program manager and the contractor that either Ada had to be used or a waiver would have to be obtained. The contractor tried to obtain a waiver based on their findings, but without actual benchmarking, their justification was deemed inadequate. The contractor truly believed Ada could not do the job, and therefore, felt they were obliged to rewrite a portion of the application in Ada to prove that a waiver was justified.

To get their proof, the contractor had one programmer rewrite approximately 10% (one critical CSU) of the application using Ada. This programmer had limited Ada experience, having only written about 5,000 LOC for a previous employer. The programmer consulted the original, experienced assembly programmer, reviewed an Ada textbook and the original software design and wrote the first Ada version of the critical CSU. The Ada code consisted of 5 pages of Ada code and comments (about 100 lines-of-Ada). It was written (based on the original design document, not a translation of the assembly code) in 2 days.

As stated above, the assembly code was written for the TI 320C15 chip, but no **Ada compiler** was targeted to this chip. Tartan (an Ada compiler vendor) had a compiler targeted to a closely-related chip and agreed to test the newly developed Ada code at their facility. They discovered that the original Ada object code was somewhat larger and slower than the assembly code. The compiler, however, optimized the code for size (making the Ada code smaller) and for speed (making the Ada code run faster). Tartan had **benchmarked** their compiler at 1.3 to 1 compiled Ada size versus compiled assembly size. (This benchmark was established by using 10,000 lines-of-assembly code written by an experienced assembly programmer compared to the same amount of Ada code written by an apprentice Ada programmer.) With some minor tweaking, the DoD contractor was amazed that ***the compiled Ada code was, in fact, faster and smaller than their assembly code for the same module!***

With these profound results, the contractor no longer pursued the Ada waiver and proceeded to use 100% Ada for the program. Within 3 weeks, the rewrite of the assembly version was completed by one Ada programmer. These 700 lines-of-Ada took about 150 hours. (This was coding time only because the documentation did not have to be rewritten.) The original development in assembly had taken approximately 1,500 hours with 600 of those hours devoted to actual coding.

This example illustrates that an Ada programmer using a **mature Ada compiler** has a very real advantage over an assembly language programmer. In this case, Ada technology advancements dispelled at least five common Ada myths by proving that:

- Ada is very suitable for real-time applications,
- Ada can be highly optimized for space,
- Ada applications are cost effective, even for relatively inexperienced programmers,
- Ada takes less time to produce, and
- Ada compilers can compete and out-perform assemblers.

This was a significant milestone in favor of using Ada for formerly exclusive assembly language applications. Until then, Ada's weakest showings were in real-time and relatively small applications where size and speed were critical. These results proved that Ada systems and capabilities had matured. The DoD contractor was thoroughly convinced of Ada's capabilities and started using it extensively. They

believe the use of Ada provides their company with a significant competitive edge. [ELAM92]

NOTE: Ada compilers can also take advantage of new chip technologies through a simple recompile. In contrast, assembly code typically requires a rewrite of the actual source code. This feature of Ada compilers is very important, given the fast rate of hardware improvement and obsolescence.

Ada Versus the Fourth Generation Language (4GL) Challenge

The IEEE defines **third generation language (3GL)** as:

> *A programming language that requires little knowledge of the computer on which the program will run, can be translated into several different machine languages, allows symbolic naming of operations and addresses, provides features designed to facilitate expression of data structures and program logic, and usually results in several machine instructions for each program statement. Examples include Ada, COBOL, Fortran, ALGOL, Pascal.* [IEEE Std.610.12.1990]

According to Daniel D. Galorath, the term **fourth generation language (4GL)** is poorly understood. All definitions claim that productivity with 4GLs is greater than with third generation languages (3GLs) (assuming the same environment, technology, and amount of reuse). The most robust and well-accepted definition of 4GL is that provided by the **Institute of Electrical and Electronics Engineers (IEEE):**

> *A computer language designed to improve the productivity achieved by higher order ("third generation languages") and, often, to make computer programming available to non-programmers. Features typically include an integrated database management system, query language, report generator, and screen definition facility...* [IEEE Std.610.12.1990]

James Martin, a software visionary, defined fourth generation language as, a *"language 10 times more powerful than COBOL."* The word *powerful* is not defined. The Microsoft Dictionary defines fourth generation language as: *"Languages designed for interacting with the programmer, often used to define languages used with relational databases."* The intent was to imply a step up from standard high level programming languages such as C, Pascal, and COBOL.

NOTE: Even the term language in 4GL may be misleading since some 4GLs have no procedural language at all.

Key Points to Consider

The following key points should be considered when comparing the use of Ada with a 4GL [HOROWITZ95]:

- 4GLs are usually touted for increasing productivity over Ada and other 3GLs. While both Ada and 4GLs allow roughly the same degree of expressiveness (Ada a little more so because of facilities such as generics, exceptions, tasking, and types), 4GLs are much more permissive — they usually do not have a strong type system, and there is no module or interface checking.
- Real productivity enhancement from 4GLs is derived from the tools supplied to support these languages. However, many of these tools are available in Ada-compatible manner, and need not be dependent on a 4GL. For example: XVT is an Ada-compatible GUI builder; bindings and support tools exist for Ada/SQL; and CORBA IDL/Ada translators exist. The availability of such tools gives DoD developers the best of both worlds: ***enhanced productivity with the safety and reliability*** of Ada.
- Remember, **development productivity is NOT the key consideration for mission-critical long-lived systems.** Support costs will far outweigh development (up to 80% of the total cost is support cost), and that effort can best be reduced by catching errors early in development, through a "*non-permissive*" engineering development approach.

Ada May Be Cheaper Than 4GLs — 4GLs May Be Cheaper Than Ada

Galorath explains that 4GLs can be cost effective when used in the proper domain and when limitations are acceptable. Ada may be less expensive when reuse, middleware, other non-developed software, or other productivity enhancements are applied. SEER-SEM *[discussed in Chapter 10, Software Tools]* was used to analyze the environment as well as the language. This model did not always conclude that Ada is cheaper or more expensive than other alternatives. As illustrated in Table 5-13, requirements definition activities are usually cheaper when 4GL methods are employed due to the ***ease*** of change and the nature of appropriate applications.

Ada May be Cheaper When...	4GLs May be Cheaper When...
No 4GL explicitly covers problem domain	The domain is appropriate
The mission must be completed exactly as specified	When limitations in implementation details can be tolerated or when using 4GL that does not have such limitations
The 4GL itself is unstable	The 4GL is available and stable
Development processes are good; development environment is advanced	Some changes/limitations can be tolerated to reduce costs
Reuse is available in Ada	No Ada reuse is available

Table 5-13 Comparison of When Ada or a 4GL Is More Cost Effective [GALORATH95]

Appropriate Domains for 4GL's

Galorath says 4GLs are usually **not** appropriate for embedded defense/ aerospace applications, computer-to-computer processing, nor hardware-intensive applications. 4GL's tend to be an option for defense user-oriented MIS systems, such as: information technology, reporting, user interfaces, and database applications. Although 4GL's can be very productive for additional reporting of existing systems (i.e., database reporting), 4GL's often require sacrifices in user requirements and design details. In analyzing 4GLs, SEER-SEM allows the general description 4th generation language or more specific identification of the language. Example 4GL and Ada scenarios show Ada reuse is the most cost effective, as illustrated in Table 5-14. (below)

ESTIMATION SCENARIO	HOUSE BUILDING SCENARIO	MAN-MONTHS	CHARACTERISTICS	NOTES
Ada reuse: Designed for reuse	Prefab house	19	Use of reusable Ada objects built to facilitate reuse	
Ada reuse: Not designed for reuse	Prepared parts must be assembled together	44	Use of existing Ada objects originally designed for mission-specific use only	The same functionality will be produced, but more rework is required
New 4GL simple processes	Prefab house	95	Use of a 4GL within its domain of appropriateness	Only applicable when the language is used for its proper purpose
New Ada with best Ada team	Build from scratch with skilled work force	361	Develop in Ada completely from scratch with a dedicated team of Ada developers	Ada tools, methods, practices in place experienced Ada staff
New Ada simple Ada processes	Build from scratch with hand tools	421	Develop in Ada completely from scratch with a dedicated team of Ada developers	This developer is not well experienced in Ada; they have no Ada tools or practices in place

NOTES: (1) Estimate is for 100 function points in each case.
(2) Estimates reflect industry norms.
(3) Detailed inputs have not been changed from their initial values.
(4) Estimates assumed 4GL was applicable in each case.

Table 5-14 When Ada Reuse is Most Cost Effective

Development Details

Figure 5-4 illustrates an example scenario, based on the examination of real-world programs, that shows full Ada reuse is cheaper than 4GL. In the real world, examine each program to determine which is cheaper.

4GL Versus Ada Size Only (Impacts Of Software Technology and Environment)

The higher the number, the higher the cost per function point. Analysis results of software size were similar between 4GLs and 3GLs. To make this analogy, the SEER-SEM model focused on effort relationships. By using the size of line expansion, insight is provided into the relative cost, with all other factors remaining constant. Table 5-15 illustrates the relative impacts of ***new*** development in Ada and in 4GL's.

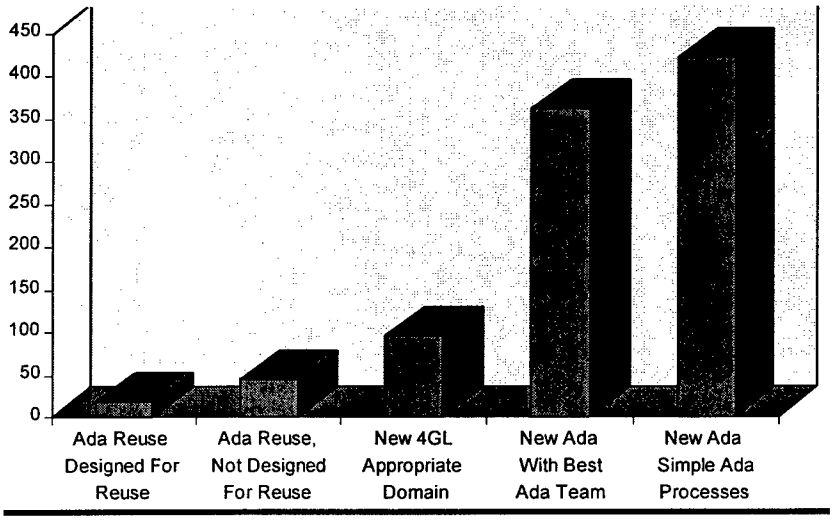


Figure 5-4 Type Programs Where Ada Reuse is Cheaper Than 4GL [GALORATH95]

	Ada (new lines per function)	4GLs (new lines per function)
Galorath (shows relative size; SEER-SEM uses more sophisticated techniques)	73	26
Jones	71	20
Reifer	72	28

Table 5-15 Comparison of Size Impacts Using Three Methods [GALORATH95]

What About Maintenance?

Vendor support of the environment can impact maintenance costs. Some 4GLs come and go and developers can be stuck with buggy, unmaintainable applications. Therefore, maintenance costs could be much lower with experienced Ada personnel than with inexperienced personnel attempting to maintain a 4GL. This risk was one of the early motivations for a standard language (i.e., Ada).

Lessons-Learned 4GL Estimation Versus Ada

- Evaluate 4GL impacts for your particular program.
- Ada may be cheaper and so may a 4GL.
- Use function-based analysis for 4GL estimation. Do not rely on simple function-to-line counts.
- Make sure definitions are in sync; the term "*4GL*" is ambiguous.
- Use 4GL's in their appropriate domain.
- Choose a 4GL that will be supported for the long haul.
- Make sure you know the 4GL's limitations and can live with the loss of design/implementation control that may occur.
- Ada reuse can have great benefits if planned for during initial development.
- Code generators for GUI, etc., can provide many 4GL benefits while working with Ada.
- Remember DoD went to Ada in the first place to promote standardization. Do not risk lack of standardization due to language until you analyze costs ***and*** benefits. [GALORATH95]

REMEMBER: 4GL's are not Silver Bullets! The key to productivity improvements are reuse, tools, practices, and experience. 4GL's can help in their own domains.

Ada LANGUAGE FEATURES

The Ada programming language contains an abundance of effective tools for expressing the solution domain so it can directly reflect your view of the problem domain. With prior languages, the solution too often had to fit the language, rather than adapting the language to the solution. Those languages only got in the way of the primary goal — solving the problem. Because Ada can directly reflect your view of the problem domain, its implementation is understandable — helping in the management of complexity.

Ada is suitable for more than simply an implementation language. As a rich source of abstract expression, Ada can also serve as a vehicle for capturing design decisions. It provides a wealth of constructs for describing primitive objects and operations, and also offers a packaging construct allowing developers to build and enforce their own abstractions. Ada has unique features not found in many production languages, such as tasking, exception handling, and packaging.

Figure 5-5 illustrates how Ada is comprised of various features that work together to support software engineering principles. [BOOCH94]

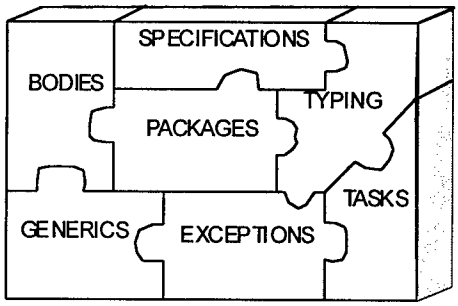


Figure 5-5 Ada Features Support Software Engineering Principles

NOTE: See Chapter 14, *Managing Software Development*, for a discussion on object-oriented development and how Ada supports that approach. For an online tutorial on Ada, access the Ada Information Clearinghouse (AdaIC) at *http://sw-eng.falls-church.va.us/AdaIC/ed-train/* under "*Ada Short Courses and Seminars*," and select "*Lovelace*."

Ada Program Unit

An Ada program is composed of one or more **program units** that can be separately compiled. Program units consist of subprograms, tasks, packages, and generic units. All program units are normally structured into two parts, the **specification** and the **body**.

The **specification** defines the information visible to the user of the program unit (the interface). The **body** contains unit implementation details that can be logically and textually hidden from the user. The specification and body can be separately compiled through separate compilation units. ***This feature is valuable in the development of large solutions. The developer can write the specification of high-level program units upfront, thus creating an enforceable design structure to the solution. Later in the development process, software implementation can be completed by independently adding and refining body units.*** Figure 5-6 (below) illustrates the concept of the two-part program unit. The specification and body support the principles of abstraction and information hiding.

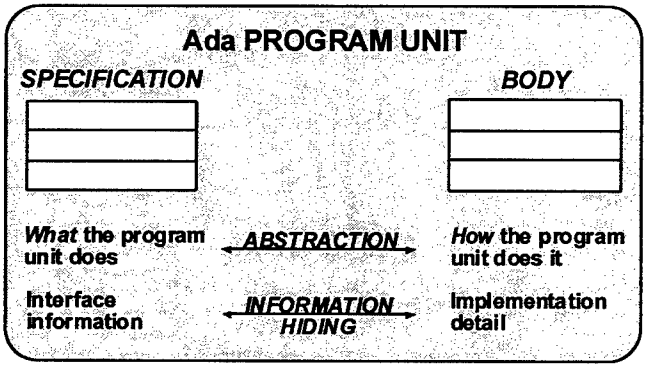


Figure 5-6 Ada Two-part Program Unit

The specification is an abstraction of the details found in the body. Implementation information is hidden from the user through the user interface.

Ada Subprograms

The Ada **subprogram** provides a means for creating abstract operations. ***Subprograms are the basic execution unit in an Ada system and can be expressed as either procedures or functions.*** Similar to program units, subprograms have two parts, a specification and a body. The **specification** (the user's interface to the subprogram) provides the name of the subprogram, its parameters, the parameter types, and whether the parameters are input, output, or both. The **body** is made up of a sequence of statements that implements the subprogram algorithm. The Ada subprogram feature supports the principles of modularity, localization, confirmability, and information hiding because subprograms can be compiled individually.

Ada's Packaging Feature

An Ada **package** contains a collection of logically-related computational resources. Packaging *encapsulates* (or puts a wall around) these resources. The package again consists of two parts, the specification and the body. The **specification** identifies the visible parts of the package and specifies which parts of the package can be employed by the user. It is not necessary for the user to understand how **body** objects and operations are implemented; thus, they are made invisible.

This structure directly supports the principles of modularity, abstraction, localization, and information hiding. Other languages have this feature, but Ada is different in that its packages *enforce and encourage* these principles. Ada language rules do not permit the user to do anything more with the package than the specification allows. Since the specification and the body can be compiled separately, the specification can be created early during software design with the body added later. Ada packages help developers control the complexity of software solutions by giving them the means with which to physically group related items into a logical picture.

Ada's packaging feature significantly reduces software costs by keeping software changes as localized as possible. It also has a positive impact on software reuse because well-designed Ada packages separate the external environment from the package, making the packages more likely to be reusable. Ada's packaging feature enables modularity by allowing data, types, and subprograms to be compiled separately. This leads to physical modularity, as small packages can be developed and tested independently. Figure 5-7 illustrates Ada's packaging construct, where the user only has access to the specification. Implementation information is hidden in the body of the package, inaccessible to the user.

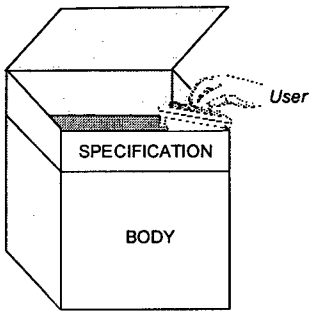


Figure 5-7 An Ada Package

Ada's Tasking Feature

Tasks are another kind of program unit. In real-time systems, many different activities often occur simultaneously. The **Tyndall Range Control System (RCS),** a tactical C2 system used to conduct air operations over the Tyndall Gulf Test Range, is an example of a real-time system implementing Ada's ability to handle multiple tasks

simultaneously. RCS operations include the real-time control of aircraft during air combat training and weapons evaluation exercises. In this case, there are multiple sensors that report data to a central facility. The real world presentation of this problem domain is a system that performs multiple tasks concurrently.

Ada's tasking feature allows the abstraction of each task to be expressed directly within the language. The RCS is closely modeled in Ada to provide realistic and accurate results to the users — the weapons controllers themselves. Ada's tasking mechanism is based on the concept of communicating sequential processes. Tasks are defined as independent, concurrent operations that communicate with each other by passing messages among themselves. In software engineering terminology, Ada is called a ***"multi-threaded language"*** as it can perform multiple tasks simultaneously within the context of a single application. [BOLEN92] Ada's tasking feature supports the principles of abstraction, modularity, localization, and confirmability.

Ada's Exception Handling Feature

In mission, safety, or security critical software systems, it is imperative that the software has the ability to recover quickly and efficiently from defects or unusual conditions. In most programming languages, an input format mistake (e.g., a divide-by-zero input) will cause the software to crash and revert back to the operating system. In reliable systems, especially real-time embedded systems, for a program to terminate abnormally could prove costly, especially where human life is at stake.

To resolve such conditions, Ada allows developers to write **exception handlers** to capture unusual occurrences often caused by latent defects and/or stressed conditions. Exceptions can either be *predefined* (e.g., a numeric overflow) or *user-defined* (e.g., an overflowing buffer condition). If an exception occurs, normal processing will cease and program control will pass to the exception handler. Exception handlers can be found at the end of a block or in the body of a subprogram, package, or task. If a handler is not present at the site of the exception, control will continue to pass up to the next invoking level, until a handler (or ultimately the operating system) is found. Figure 5-8 illustrates Ada's exception handling feature, where once an exception occurs, program control is handed over until the exception handler is found to resolve the problem. Exception handling keeps software systems from crashing under stressed conditions where undetected

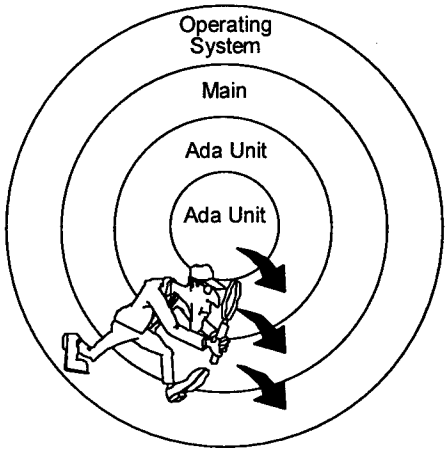


Figure 5-8 Searching for the Exception Handler

defects often surface. This feature supports the principles of modularity, confirmability, and localization.

Ada's Generics Feature

Ada was designed to handle large, complex software developments. Its special features were added to help manage ***complexity***. One of these features is the ability to individually compile specifications and bodies as separate compilation units. Another capability Ada provides is the development of **generic units** to build reusable components. A telemetry processing unit is an example where several kinds of objects must be buffered. The data types are different, but the algorithm to process the buffers is the same. To create a generic program unit, a developer simply adds a generic part to a package or subprogram specification that defines all its generic parameters. Generic package creation occurs during compilation. This Ada feature promotes, in addition to reuse, productivity and maintainability. It also supports the principles of uniformity, completeness, confirmability, modularity, localization, abstraction, and information hiding. Table 5-16 (below) summarizes Ada's program unit features.

PROGRAM UNIT	CHARACTERISTIC	APPLICATIONS
Subprogram	Sequential Action	Main Program Units Definition of Functional Control Definition of Type Operations
Package	Collection of Resources	Named Collection of Declarations Groups of Related Program Units Abstract Data Types Abstract State Machines
Task	Parallel Action	Concurrent Actions Routing Messages Controlling Resources Interrupts
Generic Unit	Template	Reusable Software Components

Table 5-16 Summary of Ada Program Unit Features

Ada Representation Specification

It is sometimes necessary to exploit the underlying features of the computer hardware system. Unlike other HOLs, where separate assembly language routines must be written to accomplish this, Ada has features that specify implementation-dependent features and data representation. An example of such a requirement might be the need to improve disk drive efficiency by writing records of only a certain length or with a given packing specification. **Representation clauses** describe how entities in the solution domain are mapped to the underlying machine to increase efficiency. When these low-level features need to be applied, abstractions about them can then be created in Ada in high-level terms. Representation specifications can be placed in the declarative part of a unit or in the specification part of Ada tasks and packages. This feature implements the principles of uniformity and abstraction.

Ada Input/Output Packages

Processors on embedded weapon systems usually do not interface with traditional I/O devices, such as printers or terminals. Instead black box interfaces are more common. To handle a range of such devices, Ada I/O is accomplished through several packages. These packages include high-level I/O, binary I/O, text I/O, and low-level I/O, supporting the principle of uniformity.

Ada's Typing Feature

In human language, the things we manipulate with verbs are called *nouns*. In Ada they are called ***"objects."*** Every object has a set of properties (called its *type* and its *subtypes*) that indicate the values of the object and the operations that are applied to that object. Ada's typing feature provides a vehicle for imposing structure on objects. **Explicit typing** is a fundamental feature of Ada that was designed to fulfill the software engineering goals of maintainability, understandability, reliability, and complexity reduction. A **type** is defined as the characterization of a set of values and a set of operations applicable to objects of a given type.

To say that Ada is a *"strongly-typed"* language means that objects of a specific type can only possess those values within that type's definition. It also means that only operations defined for that type may be applied to it. Ada's strong typing feature provides a margin of safety, since a particular type can only assume those values within the range of its definition. Strong typing also allows for the detection of more defects during compilation. This instills greater confidence that the program will be correct during its execution. Even if the typing definition is violated during execution, Ada has built-in exceptions to detect such defects programmatically. Ada's typing feature supports the principles of uniformity, completeness, and confirmability.

Ada's list of features, summarized on Table 5-17 (below), makes it a comprehensive, universally applicable, **high order language**. It was designed to be used in large, complex, frequently modified software systems that require optimum reliability, portability, reusability, and maintainability. Its vast range of useful features were designed to embrace the principles of software engineering by providing effective tools for managing the complexity of software-intensive system developments and to achieve the goals of software engineering, as illustrated on Figure 5-9 (below).

FEATURE	SOFTWARE ENGINEERING GOALS
Packages	• Allows grouping of data and related procedures for localization • Minimizes the impact of changes
Separate Compilation of Specification and Body	• Allows early definition and verification of interfaces • Documents error handling early in development • Protects local data from external modification • Supports incremental development
Typing	• Increases reliability by defining all objects and their constraints • Increases supportability by checking that they are used consistently
Generics	• Supports reusability of code
Tasks	• Facilitates concurrent processing without having to resort to assembly language and operating system calls, making development faster, maintenance easier and safer, and reducing required levels of expertise
Exception Handling	• Provides consistent mechanism for dealing with errors in data, hardware, and code

Table 5-17 Summary of Ada Feature Benefits

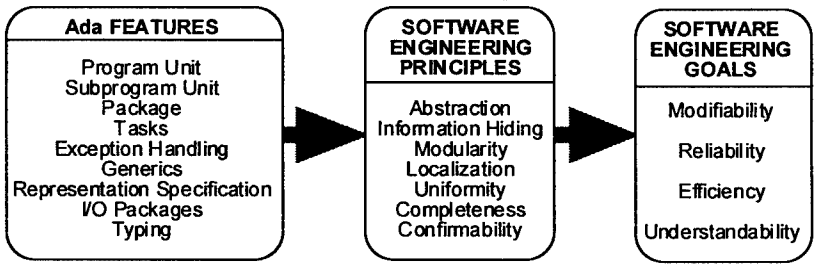


Figure 5-9 Relationships Among Ada's Features, Software Engineering Principles and Goals

Ada 95: LANGUAGE FOR THE 21ST CENTURY

In a speech to the Ada Dual-Use workshop, **Emmett Paige, Jr.**, Assistant Secretary of Defense for C³I, spoke of Ada and the future.

> *Ada has been a standardized programming language since 1983. Since its introduction, DoD and the software engineering community have benefited greatly from Ada. And with the features that Ada 9X* [now Ada 95] *promises to bring to the table, things will get even better. While we took some early heat because of lack of quality and validated compilers in the early days, the results to date show*

quantitatively that Ada makes sense, both technically and from a business point of view. [PAIGE93]

Programming language development is a continuous process. **American National Standards Institute (ANSI)** requires that language standards be revised or reaffirmed every five years. Thus, Ada engineers have developed the next generation of the Ada language, Ada 95. The Ada 95 program constituted a revision of the language, not a redesign. **Christine Anderson**, on behalf of the **Ada Joint Program Office,** conducted the revision based on ANSI and ISO procedures. The ANSI- and ISO-approved standards for Ada 95 are **ISO/OEC 8652: 1995 (E)** and **ANSI 8652: 1995**. The ISO approval makes Ada 95 the first internationally standardized, fully object-oriented programming (OOP). Ada 95 is also **a Federal Information Processing Standard (FIPS 119-1)**. Thus Ada 95 complies with DoD's direction to use commercial standards. Ada 95 will never become a military standard.

Ada 95 was designed and developed by an international process of unprecedented scale for a programming language. A Board of Distinguished Reviewers representing six different countries and comprising 28 world-renowned leaders in academia and industry provided oversight and evaluation of the immense input from the international community of users. Over 750 recommendations were received by individuals and many of the world's leading companies who were invited to submit Revision Requests. Conferences, workshops, small-group meetings, and *one-on-one* consultations were held with various segments of the Ada community, and advice was received from some of the world's finest software engineers and government technology leaders. The entire revision process took over four years to complete.

The Ada 95 design gives the language more open and extensive versatility without losing the inherent integrity and efficiency of Ada 83, the first advanced building block language to assemble a host of important features while adhering to the demands of modern software engineering practice. That is, Ada 95 keeps all the software engineering capabilities while allowing more flexibility. New features include international character sets, improved generic templates, and a set of changes that will reduce the time needed to recompile large systems. Like Ada 83, Ada 95 is a strongly typed language with full support for encapsulation and information hiding. Increased functionality allows for support of smaller, more dynamic systems. Additionally, Ada 95

provides strong support for interfaces to other languages and facilitates calls to existing subroutine libraries and OO frameworks

The additions to Ada 95, contributing to its usability, include **extended** (or tagged) **types**, a **hierarchical library** mechanism, and a greater ability to manipulate **pointers** (or references). Ada 95 incorporates the benefits of **object-oriented** languages without the insecurity wrought by a weak language foundation, as with C++. The tasking model is another area of significant Ada 95 change where the introduction of protected types allows for efficient implementation of shared data access. The benefit of speed, provided by low-level **primitives** (such as semaphores), is achieved without the risks incurred by unstructured primitives. Moreover, the use of protected types provides data-orientation in accordance with the object-oriented paradigm.

The Ada 95 standard is composed of a core, a library, and optional annexes. The library consists of a set of supplemental packages of general utility, including character handling, string handling, elementary functions, and random number generation. The annexes contain requirements to specify in detail things formerly found in Ada 83, that were not well-specified (e.g., timing requirements for immediate abort). These added specifications greatly increase performance predictability. The Ada 95 standard includes seven optional annexes covering the following topics:

- **Systems programming**. This covers a number of low-level features, such as in-line machine instructions, interrupt handling, shared variable access, and task identification.
- **Real-time**. This annex addresses various scheduling and priority issues, including setting priorities dynamically, scheduling algorithms, and entry queue protocols. It also includes detailed requirements on the abort statement for single and multiprocessor systems and a monotonic time package (distinct from the Calendar which might go backwards due to time-zone or daylight savings changes).
- **Distributed systems**. The core language introduces the idea of a partition, whereby one coherent "*program*" is distributed over a number of partitions, each with its own environment task. This annex defines two forms of partitions and inter-partition communications using statically- and dynamically-bound remote subprogram calls.
- **Information systems**. The core language extends fixed-point types to include basic support for decimal types. This annex defines a number of packages providing detailed facilities for manipulating

decimal values and conversion to external format using picture strings.

- **Numerics**. This annex addresses the special needs of the numeric community. One significant change is the basis for *model numbers*. These are no longer described in the core language but in this annex. Moreover, model numbers in Ada 95 are essentially what were called "*safe numbers*" in Ada 83. The old model numbers and the term "*safe numbers*" have been abandoned. Having both *safe* and *model* numbers did not bring benefits commensurate with the complexity and confusion they introduced. This annex also includes packages for manipulating complex numbers.
- **Safety and security**. This annex addresses restrictions on the use of the language and requirements of compilation systems used in safety-critical and related applications where security is vital.
- **Language interfaces**. This annex defines additional facilities for communication with applications in other languages, especially C, COBOL, and Fortran.

Christine Anderson, former program manager for Ada 95 and coeditor of the revised standard reference manual, explained: *"No other language has ever been created following written requirements refined by the world's best in computer programming and software development. Ada 95 is the culmination of these efforts, thereby delivering the most viable, cost-effective language for the development of long-term software solutions."* [ANDERSON94]

The importance of the Ada 95 program to DoD was expressed by **Emmett Paige, Jr**. Upon taking office as Assistant Secretary of Defense for C3I, Paige sent an electronic mail message to the Deputy Assistant Secretary of Defense for Information Systems, the Air Force Deputy Assistant Secretary for Communications, Computers, and Logistics, and the head of DISA. In it he told them:

> *Ada is not dead, Ada is alive and doing well and is needed as much as ever, despite the views of some senior folks on the interview and speaking circuit. We cannot lock Ada [95] in a straight jacket with long drawn out extensive procedures for making changes. Please tell me what I must do to get the show on the road and rejuvenate the Ada effort, to include opening up the reuse repositories to industry and academia without any inhibitions. We must open the gates if we are to be successful in encouraging the use of Ada in the private sector.* [PAIGE93]

Ada IMPLEMENTATION

Ada was designed to support the development of major software-intensive weapons and MIS systems with extended life cycles. Typically, these applications are developed by large staffs divided among multiple contractors and/or organizations. The following guidance is based on lessons-learned from a range of major Ada development efforts. Regardless of the development stage of your program, this information will help you implement Ada, along with sound software engineering practices, into your development process. Never forget, ***Ada and software engineering go hand-in-hand.***

Transitioning to Ada 95

Change is seldom easy or instant; but in the case of moving to Ada 95, the benefits are worth the effort and programs are already leading the way. Two documents are available to help you make the transition from the AdaIC in paper copy or downloadable from their Internet host *[http://sw-eng.falls-church.va.us]*.

- **Ada 95 Adoption Handbook.** The *Handbook* explains the opportunities, issues, and answers involved in adopting Ada 95;
- **Ada 95 Transition Planning Guide.** The *Planning Guide* helps managers who have decided to adopt Ada 95 create a detailed transition plan.

These documents cover a wide range of topics — among them are issues related to program planning; tools and environments; upward compatibility; technology transfer; software-development methodologies; commercial-off-the-shelf products, legacy software, and multi-language development; and the adoption process and personnel. Other publications available to guide your transition to Ada 95, all available online from the AdaIC, include

- **Ada 95 Adoption Handbook**, (listed above) March 30, 1995,
- **Ada 95 Transition Planning Guide**, (listed above) September 30, 1994,
- **Ada 95 Language Reference Manual,**
- **Ada 95 Rationale,**
- **Ada 95 Annotated Ada Reference Manual,**
- **Ada Compatibility Guide,** Version 6.0, January 1995,
- **An Overview of Ada 95,**

- **Contrasts: Ada 95 and C++,**
- **How to Program in Ada 95 Using Ada 83,** and
- **Multiple Inheritance in Ada 95**

Table 5-18 provides a list for determining whether it is too early, the time is right, or you are past due in transitioning to Ada 95.

IT IS TOO EARLY IF..	IT IS TIME TO TRANSITION IF...	IT IS PAST DUE IF...
Your staff is uninformed	Your staff is informed, eager, and ready to adopt	Staff is trying to transition without management support
You are unable to estimate the impact on schedule and budget	The impact on program planning is understood	Other sister programs are using Ada 95 as part of their cost and schedule risk reduction efforts
Ada 95 tools have not been evaluated	Tools are available and mature (i.e., *production ready*)	You are spending funds on support of older tool rather than for upgrades
Your Ada 83 software has not been evaluated for upward compatibility	Your Ada 83 software has been made upward compatible	You are spending money on workarounds for features found in Ada 95
Your developer has no training or mentoring programs	Initial training/mentoring is available and acquired	Most other sister programs have already trained their staffs
Software methodology use is inconsistent	Impact of changing methods has been assessed	You are suffering productivity loss from use of older software development methods
You have not yet determined if your program schedule is inflexible	You are at the start of your program or a major support enhancement	You are in the middle of an existing program with strict deadlines

Table 5-18 How to Know When to Transition to Ada 95

Adopt An Incremental Transition Strategy

When considering how to make the transition, the *Planning Guide* advises an incremental transition strategy, in which a few new Ada 95 features are introduced at first, and others are adopted later. This enables you to shorten the learning curve and lessen the impact on your program's budget and schedule. If you adopt Ada 95 to exploit new technologies such as object-oriented programming, then making sure your team ***learns*** to use these new technologies will be a major risk item that can impact cost and schedule. These underlying technology shifts are present anytime there are changes in programming languages, tools, or development methodologies.

Write Ada 95-Compatible Code in Ada 83

One step in your transition can be to have your developer start writing Ada 95 code in Ada 83. *[See in "How Can I Write Ada-95-Compatible Code in Ada 83?", AdaICNEWS, Spring 1995.]* Ada compiler vendors have released versions of their Ada 95 compilers. Many of these compilers process both Ada 83 and Ada 95 code and provide the user with a switch to choose the appropriate mode. A switch-selectable Ada 83/ Ada 95 compiler reduces program risk, because your software developer can use one tool to move back and forth from Ada 83 mode to Ada 95 mode until they completely transition to Ada 95.

NOTE: See "*Choosing the Appropriate Ada 95 Compiler*" under "*Ada Technology Considerations*" below.

Remember the Human Factor

Remember that ***people*** are involved in the adoption process. Change can be difficult for any team. You should make sure that the adoption of Ada 95 is a participatory process and that your software developer is prepared for, and actively involved in, transition planning, as well as, the implementation phases. While this may seem to be a *"soft"* issue compared with the availability of compilers or the performance of Ada run-time kernels, one of the most important lessons of previous technology adoptions is that social and environmental changes are often the most important ones, with the greatest impact on transition success.

When transitioning from another language to Ada 95, the paradigm and mindset shifts are greater than they are when moving from Ada 83 to Ada 95. Each programming language brings with it a large collection of techniques and methods that reflect *"how software ought to be built"* using that language. Changing from one language to another is not as simple as switching from one make of automobile to another. Rather, it is more akin to switching from driving a car to piloting a jet. Therefore, you should take the following steps to make sure your software developer successfully adopts the Ada 95 culture:

- **Support technology transfer** efforts that educate developers, not only to the syntax of the new language, but also to the way it should be used (its culture).
- **Provide mentors** with Ada experience who can help guide the development team.

- **Hire experienced Ada personnel** and seed them on the development team as group leaders who are responsible for showing the entire team how to create software *"with an Ada mindset."*

Ada TECHNOLOGY CONSIDERATIONS

There are certain technology considerations you should evaluate when implementing your Ada development. These apply to all Ada programs, whether yours is a new start, an on-going development, or one in PDSS. These considerations include the following.

Ada Compilers

A **compiler** is a software tool that translates a higher order language, such as Ada, into machine language. It usually first generates assembly language, then translates assembly language into machine language. An Ada compiler includes the compiler, program library system, linker/loader, run-time system, and debugger. Compilers not only support software development products and methodology, but program-specific procedures as well.

Compiler Selection

Compiler selection is not to be taken lightly, the method for which depends on how you structure your RFP. You may choose to leave its selection up to the winning contractor, require its selection to be included in offerors' proposals, or stipulate the use of the compiler you have pre-selected. In any case, you must take the time to educate yourself on available alternative choices to determine what is best for your program. Your compiler selection process starts with your **Acquisition Plan** *[discussed in Chapter 6]* that establishes program requirements, budget, personnel, and schedule. Criteria for compiler selection are program-specific. (For example, execution time is not as critical in an MIS as it is for weapon systems software.) Benchmarks, checklists, and interviews with vendors and users should be employed as a means for assessing whether candidate compilers meet your requirements. Your evaluation should also include a limited number of alternative compilers with the potential for fulfilling your needs. *[Contact the Ada Information Clearinghouse (discussed below) for their suggestions on which compilers to consider for your specific domain.]*

The evaluation and selection of an Ada compilation system is a serious, critical process that should be addressed in your **Risk Management Plan**. Careful assessment and selection of an Ada compiler package can greatly decrease overall program risk, cost, and schedule overruns. Be aware that evaluation and selection applies to the ***entire*** software development package, not just the compiler. Your selection process must include identification of key criteria and testing of candidate compilation systems against those criteria. There is no single test suite or set checklist that suffices for every program. However, there are several **benchmarks** *[discussed below]* and **test suites** that can be used to evaluate compiler implementation after program requirements have been identified. *[Refer to SEI's Ada Adoption Handbook: Compiler Evaluation and Selection, Version 1.0]* [WEIDERMAN89[1]]

NOTE: Normally, you will have contractors propose a compiler; however, you should make this a specific evaluation criterion during source selection.

Compiler Maturity

The maturation of Ada compilers and software engineering environments, like fine wine, takes time. Using immature tools can negatively impact productivity if your software developers have to concurrently develop and maintain new applications, new tools, and new hardware. You should consider the extent to which a compiler has been used in production work when developing program milestones. ***When possible, select a widely used, mature compiler.*** *[The STSC is a good source for information on compiler maturity.]*

Compiler Validation

All software development efforts are required to use a validated Ada compiler. Not only must you select a **validated compiler**, you must also verify that the compiler(s) you select is validated against the most current version of the validation test suite.

You should schedule procurement of your compiler to correspond with program start. During the program life cycle, as technology improves, you may want to upgrade operating systems, compilers, editors, and CASE tools to the latest version. However, it is your responsibility to wisely control upgrades because even minor upgrades can have serious impacts on program cost and schedule.

Compiler Evaluation

Ada compilers can be evaluated using the **Ada Compiler Evaluation System (AECS)**. The goal of formal evaluation is to provide comparative compiler performance data to vendors, procurers, and users of Ada products. These data give vendors a baseline for compiler performance improvement, give procurers implementation and configuration information for selecting the best compiler for their needs, and give users the means to identify the language features that are best to use or avoid for their particular requirements.

A word of caution about compiler evaluation can be learned from a **1912th Computer Systems Group** team at **Air Combat Command**. They were tasked to port a highly visible C2 system from one hardware suite to another and assumed that because the system was developed in Ada, porting would be a relatively easy task. Unfortunately, they found that unless the designers and coders of a system use sound software engineering practices, the porting task can be extremely risky.

The subject system's source and target hardware platforms were both produced by the same company, but with different operating systems. Once ported, the target compiler produced markedly different results than the original compiler. Although both the old and new compilers were made by the same company, they did not implement Ada in the same way. This suggested to the team, that while an Ada validation test suite [Ada Compiler Validation Capability (ACVC)] was used to verify that the compilers conformed with the Ada language standard, that particular test suite did not determine how well the standard was implemented. They found that a creative solution to compiler inconsistencies is for the developer to be required to successfully compile the software on three different compilers. Also, use of the ACES test suite would have helped to better identify potential compiler problems. [GAETANO92]

Compiler Benchmarks

Because all compilers are not alike, **benchmarks** give techniques and application examples with which to compare Ada compiler performance with other Ada compilers or with other language compilers. The results generated through execution of a benchmark program can also help test the characteristics of candidate compilers. Available compiler benchmarks include:

- The **Ada Compiler Evaluation System (ACES)** benchmark measures performance, emphasizing execution speed, code size, compilation speed, system capacities, and capabilities of the system's symbolic debugger, diagnostic messages, and library system.
- The **Performance Interface Working Group (PIWG)** benchmark measures execution speed.
- The **HARTSTONE** benchmark measures a system's ability to handle HArd Real-Time (HART) applications. [WEIDERMAN89[2]] [ALLEN92]

For some medium to large programs, you may not be able to rely on publicly available resources and may have to develop your own benchmarks that reflect your specific needs. If you are developing your own benchmarks, consider the following:

- Benchmarks must represent and test system requirements and the environment selected;
- Benchmarks must be a part of a planned, totally integrated and supported test suite;
- Benchmarks must be maintained throughout the software life cycle; and
- Benchmarks and benchmark requirements must be suitable for inclusion in a contract.

[Contact the SEI or a reference such as Camp's Benchmarking for information about new benchmarks for Ada applications.] [CAMP89] *[More information about benchmarks can be also obtained through the Ada Information Clearinghouse, discussed below.]*

Choosing the Appropriate Ada 95 Compiler

Ada 95 compilers provide developers with new opportunities; however, these can lead to new risks. To reduce the risk associated with using new Ada 95 compilers, you, your development staff, and/or your contractors must take steps to assess compiler maturity and usability on upcoming programs. The most important techniques are:

- Benchmarking the compiler using a set of standard tests;
- Supplementing the standard benchmark tests with program-specific benchmarks reflecting high-risk areas of program concern (e.g., speed of compilation, execution speed of real-time features, size of generated code for generics, etc.); and
- Running a pilot program that measures the compiler's usability under typical run-time conditions.

Since the publication of the new Ada 95 standard, nine major vendors have announced they plan to validate and market Ada 95 compilers. Currently these vendors produce over 60% of the 839 current Ada 83 compilers. Table 5-19 lists these vendors and a point of contact.

VENDOR	POINT OF CONTACT
Intermetrics, Inc. 733 Concord Avenue Cambridge, MA 02138	Bill Zimmerman Phone: (617) 661-1840 E-mail: *billz@inmet.com*
OC Systems 9900 Lee Highway, Suite 270 Fairfax, VA 22030	Oliver E. Cole Phone: (703) 59-8160 E-mail: *info@ocsystems.com*
Rational Software Corporation 2800 San Tomas Exprssway Santa Clara, CA 95051-0951	Kevin Knix Phone: (408) 496-3600
R.R. Software, Inc. P.O. Box 1512 Madison, WI 53701	Ian Goldberg Phone: (608) 245-0375 E-mail: ian@software.com
Silicon Graphics 2011 North Shoreline Boulevard Mountain View, CA 94043	Dave McAllister Michael Stebbans Phone: (800) 833-0085
Sun Microsystems, Inc. Sun Pro, Inc. 2550 Garcia Avenue MS: UMPKO3-205 Mountain View, CA 94043-1100	Carole Amos Phone: (415) 688-9424, (415) 968-639 E-mail: *carole.amos@eng.sun.com*
Tartan, Inc. 300 Oxford Drive Pittsburgh, PA 15146	Wayne Lieberman Phone: (412) 856-3600 E-mail: *lieberman@tartan.com*
Thomson Software Products 10251 Vista Sorrento Parkway Suite 300 San Diego, CA 92121	Marianne Worley Phone: (800) 833-0085 weburl: *http://www.thomsoft.com*
Unisys Corporation 506 Highway 85 North Niceville, FL 32578	Joseph Kovach Phone: (904) 678-4217

Table 5-19 List of Current Ada 95 Compiler Vendors

Ada Interface Standards

Ada must be able to ***interface*** with the resources end users need to control. Interfaces to these resources have been standardized to ensure Ada's interoperability, portability, and reuse. Commonly used resources with which Ada must interface are:

- **Operating Systems**. (e.g., POSIX)
- **Databases**. [e.g., Structured Query Language (SQL™), Information Resource Dictionary System (IRDS)]

- **User interfaces**. (e.g., XWindows, Motif, Open Look)
- **Graphics**. [e.g., Graphical Kernel Systems (GKS), Programmers Hierarchical Interactive Graphics System (PHIGS)]
- **Networked resources**. [e.g., Government Open Systems Interconnection Profile (GOSIP), X.25, X.400, X.500]
- **Hardware**. (e.g., 1553 data bus)

Both DoD and commercial standards are important to software development. However, conformance with commercial standards encourages **open system environments (OSEs)** needed to attain functionality and to provide interoperability, portability, and scalability of software applications across networks of heterogeneous hardware and software. *[Consult the Federal Information Processing Standards (FIPS) (discussed in Chapter 2, DoD Software Acquisition Environment) for MIS technology requirements.]*

Secondary standards provide interfaces to computational resources. These standards are used for computing mathematical functions, rational numbers, statistical functions, and decimal arithmetic. Secondary standards support reuse, interoperability, portability, and are commercially available to support Ada software development.

Ada Bindings

For an interface standard to become an *"international standard"* it must be written in an abstract specification (e.g., VDM, SL, Z). (Until recently, interface standards could be written in an application language as an abstraction specification; e.g., POSIX's basic services were written in C.) For an application language to use the international standard, a language binding to the interface standard is required. **Bindings** allow software applications to interface with other software products conforming to the same standard. Figure 5-10 illustrates how Ada bindings are used to connect Ada applications with computer resources through **interface standards**. Development and standardization of bindings are the mission of national and international organizations such as ANSI and the International Standards Organization (ISO).

Ada bindings to an interface standard are usually in the form of an **Ada package specification**. To use the Ada binding, an implementation in the Ada package body is required. The Ada package feature provides an excellent vehicle for supporting interface standards. Thus, Ada is an ideal facilitator for OSEs. To support DoD's migration to

CHAPTER 5 Ada: The Enabling Technology

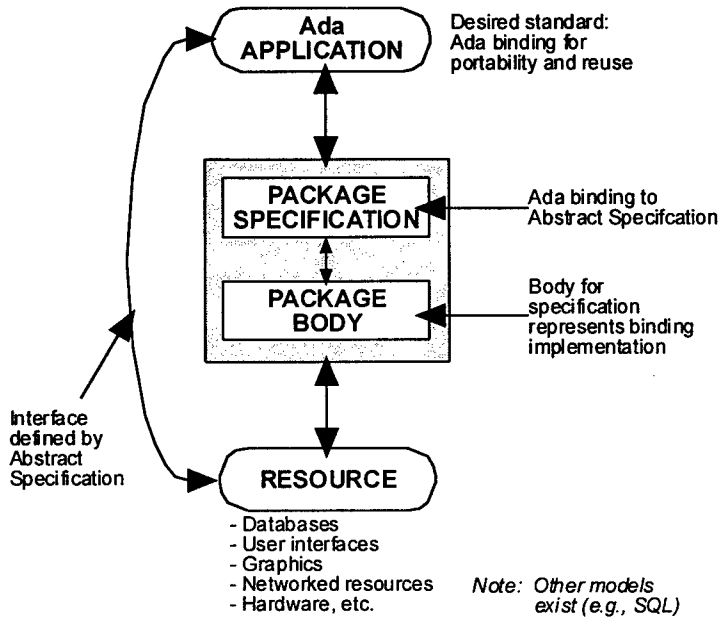


Figure 5-10 Ada Bindings

OSEs, commercially available Ada bindings support the following interface standards:

- Ada Semantic Interface Specification (ASIS),
- Generic Package of Elementary Functions (GPEF),
- Generic Package of Primitive Functions (GPPF),
- GKS, PHIGS, POSIX, SQL,™
- Transmission Control Protocol/Internet Protocol (TCP/IP),
- XWindows, Motif, Open Look, X.25, X.400, and X.500, and
- Microsoft Windows.

[The Ada Information Clearinghouse Report, Available Ada Bindings, updated quarterly, gives a brief description of the standard and a point of contact for each commercial product supporting an Ada binding.]

Operating Systems

An **operating system** is the software or firmware master control structure closest to the computer hardware that acts as a combination of scheduler and traffic cop (e.g., MS-DOS, UNIX, VMS). Operating systems reside on host and target processors. **Host processors** are usually used to develop software. **Target processors** run or execute the developed software. Most MISs and land-based military applications develop and implement software on the same processor or the same family of processors. When software is developed and executed on the same type processor, the operating system has an integrated **run-time environment (RTE)**. The application (compiled program code) is bound to the operating system at link time and the entire entity becomes the application.

Commonly in weapon system applications, the target processor is not the same as the host processor. For these systems, all system services are provided by the application operating system, the RTE. Ada has been very effective for applications with time-critical processing requirements on RTEs.

Validated Ada products must conform to the ISO/ANSI standard, but this standard does not address run-time issues. Due to Ada's philosophy and structure, there is usually a difference between Ada's run-time requirements and other languages that use more conventional operating system products. Most Ada run-time products are based on proprietary commercial operating systems and the bindings are usually written in a language other than Ada.

Databases

SQL™ is an interface standard for use with a standardized language in **relational database management systems (RDBMSs)**. The major problem with SQL-compliant database programs is the use of extensions which tend to degrade portability. The **SQL Ada Module Description Language (SAMeDL)** is a binding from Ada to SQL-based databases. SAMeDL is designed to facilitate the construction of Ada database applications that conform to the **SQL Ada Module Extension (SAME)** architecture. It extends the module language defined by the ANSI SQL standard to better fit the needs of Ada. *[Descriptions of SAMeDL and SAME can be found in SEI technical report CMU/SEI-90-TR-25.]*

Graphics

The **Graphics Standard Interface Standard (GSIS)** is a set of standards essential to the acquisition of next-generation computing systems for DoD. The GSIS aids in addressing a wide range of functionality for various levels of military graphics applications. An **interactive graphics system** is a set of hardware and software that works together to provide the following services and physical devices:

- It accepts human input (e.g., via a mouse, keyboard, buttons, touch screen) and presents it to the application program through an accepted protocol (procedural interface) in an accepted presentation format; and
- It presents data and information (e.g., display formats and images on cathode ray tubes) to users through a protocol adhered to the application program and the graphics system as the provider of graphics services.

Windowing Environment

The **XWindows** system is a hardware-independent and operating system-independent graphics standard designed to operate over a network or within a stand-alone machine. The libraries (Xlib and XT) are basic in scope and provide a communication protocol for XWindows which are in turn used by higher-level tool kits (e.g., Motif, Open Look) to facilitate the writing of user interfaces. *[Ada bindings to the Xlib and XT XWindow interface are available free from the STARS repository, discussed in Chapter 4, Engineering Software-Intensive Systems. In addition, there are products which provide various levels of interfaces to Microsoft Windows. These include Screen Machine from Objective Interface Systems, Reston, Virginia and Janus/Ada Windows Toolkit from RR Software, Inc., Madison, Wisconsin.]*

Ada Run-time Efficiency

Weapon system software must respond quickly and efficiently to external stimuli, and MIS applications must accomplish the most work possible on available resources. You should, therefore, carefully evaluate run-time efficiency. **Run-time efficiency** can be assessed through compiler evaluation, benchmarking, and prototyping.

NOTE: See Chapter 10, *Software Tools*, for a discussion on Ada Tools, Methods, and Support Centers. Also see Chapter 9, *Reuse*, for a discussion on Ada Reuse.

Ada AND YOUR PROGRAM

The size and complexity of major software-intensive weapon systems, MISs, and their upgrades present engineering and management challenges that take dedicated effort to comprehend and remedy. You need to integrate systems engineering, software engineering, standards, development methodologies, life cycle management, metrics, open systems, architecture, risk management, and reuse in your management process to ensure the software you develop is of the highest quality, on time, within budget. ***Simply using Ada does not guarantee quality software!*** What Ada can do for you is make the management activities you must perform *[as outlined in Chapter 4, Engineering Software-Intensive Systems, and throughout these Guidelines]* ***easier to institute and more controllable.***

Ada and Your New-Start Program

If your program is a **major new-start software development**, you need to follow the guidance in DoDI 5000.2 and DoDD 3405.1 concerning Ada. However, before you stipulate in your RFP that all your code must be developed in Ada, ***explore the possibilities of how much of your functionality can be accomplished through COTS and reuse***. The 100% custom developments of the past are not economical. Efforts such as domain engineering, which capitalizes on reuse and COTS, should be considered as a cost effective alternative. *[See Chapter 4, Engineering Software-Intensive Systems, and Chapter 9, Reuse, for more information on Domain Engineering.]* Another alternative might be to scale back on initial system requirements, and incrementally add functionality as the system evolves. *[This incremental strategy is discussed in Chapter 3, System Life Cycle and Methodologies.]* Of course, the decision you make will be based on the system requirements, acquisition environment, and the life cycle requirements.

NOTE: During source selection, look for contractors who can demonstrate an executable Ada architecture using UNAS (or equivalent) and who have a robust software

development environment like the Rational Environment™ (or equivalent). See Chapter 10, *Software Tools*, for a discussion on these two systems.

Ada and Your On-going Program

If you are managing a **major on-going software development** that is not implementing Ada, you should consider your options. For the foreseeable future, DoD is committed to Ada. All software maintenance organizations will support Ada products. For your program to be successful, it must be ***maintainable***. *"Other than Ada"* code represents the legacy of high-cost software we can no longer afford to support. You should conduct a life cycle cost analysis of the software products you are building to determine whether changing languages is financially practical. You will find developing software in Ada is always a good business decision in the *long-term*. Changing to Ada midstream may not always be economically feasible in the short-term, given today's budgets and DoD's incremental funding cycle. Regrettably, it is not always possible to spend a little more today to save a lot tomorrow.

However, if your software architecture is properly structured, those modules that are stand alone and yet to be built ***should be coded in Ada***. The life cycle cost savings you are going to gain by doing so will greatly improve your program's chances for success. Using Ada as your primary development language, combined with modern software engineering methods, will result in code that is reliable, supportable, and reusable.

Ada and Your PDSS Program

If you are managing the **support of non-Ada legacy software**, you have several options that can reduce your program costs and improve the quality of the product you are supporting. You must first conduct a life cycle cost analysis to determine whether it is best to continue patching your existing code, to start from scratch with a new Ada development, or to re-engineer your existing code to Ada. *[Consult the STSC for evaluations of life cycle cost models best for your program.]* If you decide to continue to maintain your non-Ada code, ***major modifications should be coded in Ada and any localized, stand-alone changes should be performed in Ada.***

NOTE: See Chapter 11, *Software Support*, for an in-depth discussion on re-engineering legacy software to Ada.

Ada Upgrade Opportunities

Because our future software support environments will depend mainly on Ada, it is smart for organizations to embrace opportunities to use Ada when upgrading existing resources. The benefits of using Ada for upgrades and enhancements to existing systems include ease of maintenance, reusable libraries, tool variety, and programmer productivity. When upgrading a current system, you should address the following technology considerations.

Mixing Ada with Other Languages

Upgrades to existing systems are currently more common in DoD than new-start programs. A major challenge to program managers involved in these upgrades is to develop viable strategies for ***inserting Ada into existing systems***. Alternative strategies can be evaluated by developing working hybrid models of the new structure. Some problems related to inserting Ada into existing systems are data handling, scheduling, and program libraries. It is sometimes more feasible, from a technical and operational perspective, to add processing elements that are pure Ada and to place greater emphasis on interchanging data. This alternative is illustrated in Figure 5-11 (below) where Ada modules and non-Ada modules have separate compilation units which are then linked to produce the application. When using this approach, you must thoroughly address architectural concerns such as data, program libraries, degraded mode reconfiguration, and networking. Another alternative is **co-processing** which is now progressing from an emerging technology to a practical solution.

Porting with Ada

Porting is the ability to use data and software on different computer hardware systems. Over the life of a software-intensive system, the host hardware environment frequently changes. These changes can be caused by hardware upgrades, modernization, transition of support management responsibility, or diminishing (or loss of) host processor vendor support. If these changes occur, application source code and program-developed software tools are candidates for porting. As with reuse, portability must be considered upfront when developing software

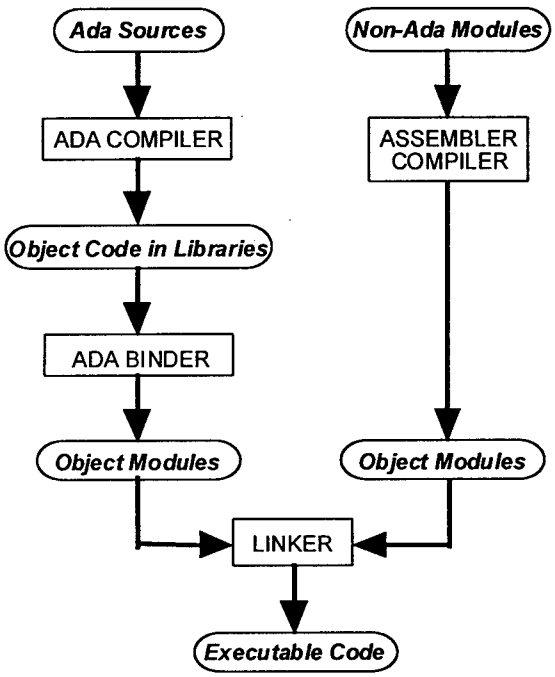


Figure 5-11 Linking Ada to Non-Ada Modules

and tools. Although Ada applications are generally more portable from one environment to another than other languages, you must take an active role in ensuring your software ports easily.

Requirements and Design Impacts

When translating or converting existing code to Ada, the operating system must be an Ada run-time operating system or be compatible with the Ada run-time library and the Ada **run-time environment (RTE)**. If the operating system is incompatible with the Ada RTE, major changes must be made to the application and/or operating system's design. If this type change is not identified and planned for in the very early stages of transition, through modifications to requirements and design, you may experience a negative return on investment.

ADDRESSING Ada IN THE RFP

It is imperative you select a high quality software organization that has demonstrated experience in exploiting the advantages of the Ada language. There are a growing number of software developers who have produced tools that enable them to develop full-scale operating models and prototypes of a proposed system's capability (e.g., **UNAS** marketed by **Rational Environment™**). This means you do not need to rely on *"paper analysis"* to evaluate the effectiveness of proposed designs. When you have offerors (identified through market and pre-award surveys) who can rapidly construct and demonstrate **working Ada models** based on an architectural skeleton, ***you should require such demonstrations as part of their proposal.*** Another approach when requiring these demonstrations is to carry a minimum of two contractors to PDR. This approach can be used where contractors need to develop engineering tools to maintain system control and data integration in the design and coding process. In this case, you can extend the PDR for some reasonable period to compensate for the extra development. With the ability to evaluate an operating model of the proposed application, the modest additional cost and schedule time pays for itself through beneficial risk reduction.. [BAKER92]

CAUTION: The first Ada program performed by a developer can be problematic; therefore, when selecting your contractor, the more Ada experience the better. Insist that contractors describe their Ada experience and associated Ada software development environment as part of their proposal. Ensure that this is a major, if not THE major, risk consideration in source selection. ESC has prepared a supplement to the Software Capability Evaluation *[discussed in Chapter 7, Software Development Maturity]* **specifically addressing the maturity of a contractor's Ada capability.** *[Also see Volume 2, Appendix M, Tab 1.]*

Your RFP should require that bidders furnish historical data (size/ time/effort) on 3-4 completed programs in the same domain and of comparable size and complexity. These data should then be used to assess the bidder's productivity on those historical programs. RFPs should also require that offerors provide an **estimate of software quality** (expected defects remaining) and **reliability** *[mean-time-*

to-defect (MTTD)] at delivery, along with an estimate of the additional time required to make the software 99.9% free of expected defects.

Assessing the software development capability of offerors provides a snapshot of their past process implementation, current process activities, and future process potential. Although these evaluations are an excellent way to assess the capability of potential contractors, most of the issues addressed are not specific to any particular language. An additional set of questions must be asked concerning Ada. There are six **key process areas (KPAs)** in the **Software Capability Evaluation (SCE)** that can be tailored to include the assessment of the offeror's Ada capability. The information gathered under *"training program"* and *"software program planning"* KPAs will show how well an offeror plans for Ada programs. *"Software tracking and oversight," "software product engineering," "organization process definition,"* and *"software quality assurance"* KPAs can be used to determine how well an offeror performs on Ada programs. *[Volume 2, Appendix M, Tab 1 contains KPAs from the SEI CMM and questions for a SCE team to use in assessing a contractor's Ada capabilities.]*

Ada Waivers

There might be times when a language other than Ada is proposed. ***Such proposals should be required to provide strong justification that the overall life cycle costs will be less than with the use of Ada.*** If a waiver (or exception) is considered, it must be obtained in accordance with DoD 5000.2-R and DoDD 3405.1. Some of the lessons-learned from life cycle cost estimating efforts show factors which must be considered when submitting an Ada waiver request. These factors include:

Factor: ***The overall cost to train programmers in Ada is less than normally projected.***

Rationale: (1) Only a core group of programmers are kept from program to program. The rest are hired as needed or transferred from other parts of the company and must be trained anyway.

(2) The differences between C, or any other high-level language (HOL), and Ada are not difficult enough to prohibit programmers from learning to program in Ada at an acceptable rate. Programmers are normally exposed to several HOLs during their

education and are able to grasp Ada concepts and constructs.

Factor: ***The overall cost to purchase Ada software engineering environments (SEEs) is less than normally projected.***

Rationale: (1) Whether Ada-oriented or not, companies always upgrade and acquire new tools and SEEs.

(2) SEE costs can be (and are often) shared between programs (including future programs).

Factor: ***Language proficiency (i.e., programmer productivity) is language-independent and increases during the software life cycle. This increase must be factored into life cycle cost analyses.***

Rationale: (1) Proficiency evolves from *low* to *high* as the team works with the language.

(2) Every design change (e.g., block change cycle update) results in an increase in proficiency. After the first change, proficiency should move from *low* to *medium*; after the second change, proficiency should move from *medium* to *high*.

(3) Maintenance costs should be calculated at a *low* proficiency for the first year or two, at *medium* for the third to fourth year, and at *high* for the rest of the life cycle.

Factor: ***Toolset and SEE proficiency increases during the software life cycle.***

Rationale: Increasing tool proficiency must be factored into the total life cycle cost analyses. This can be done in a manner similar to language proficiency.

Ada: THE LANGUAGE OF CHOICE

Rear Admiral Alfred T. Mahan authored a book in 1890, The Influence of Sea Power on History, that revolutionized naval strategic planning worldwide. In 1908 he made a statement that could be echoed today to express why we use Ada.

> *Whatever the system adopted, it must aim above all at perfect efficiency in military action; and the nearer it approaches to this ideal the better it is.* [MAHAN08]

Ada increases the chances for success of every program manager of a major software-intensive system in DoD. If you are managing a new start program, you are using Ada and are equipped for success. Never forget, Ada and software engineering discipline go *hand-in-hand.* Only through a disciplined approach will you produce good Ada software. If you are managing an on-going non-Ada program, consider the total life cycle benefits of transitioning to Ada. Whatever your choice about your already developed code, any major new localized or stand-alone modules must be developed in Ada. If you are managing a program in PDSS phase, consider the economic life cycle benefits of re-engineering your legacy software to Ada. Be aware, all major modifications and enhancements to existing code must be performed in Ada. Figure 5-12 summarizes the benefits of Ada language features.

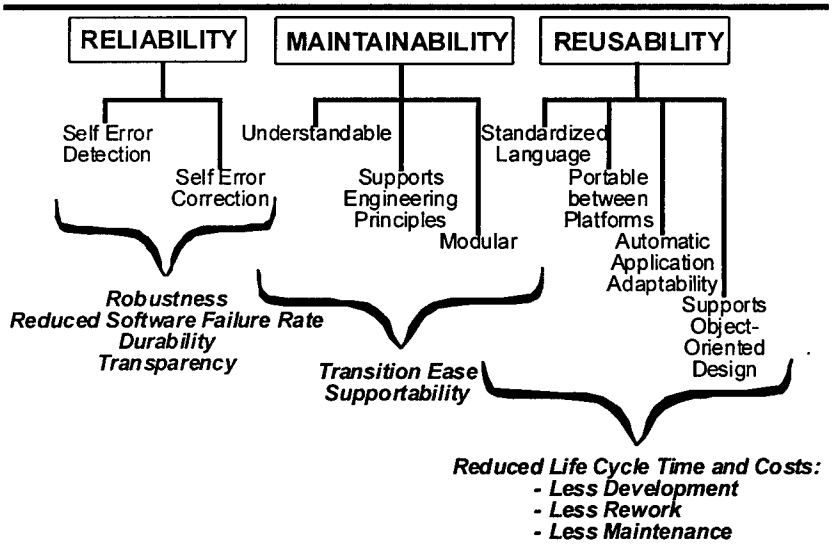


Figure 5-12 Summary of Ada Feature Benefits

CHAPTER 5 Ada: The Enabling Technology

Approaching perfect efficiency in today's military environment is the ability to incur the greatest damage, while saving the most lives. During the **Gulf War**, Ada software proved it can deliver on both counts. This is one reason why it is smart to use Ada. It is not, however, smart to just use Ada to destroy enemies and save lives. It is also smart to use Ada for any large, complex software-intensive system with a critical mission. Ada's ability to support the principles of software engineering makes it an enabling technology that can produce reliable, supportable, and economical software. As **Lieutenant General Edmonds** emphasized,

> *We cannot forge ahead without a disciplined, reasoned approach to software development...Ada isn't the perfect language. There never was such a thing, and I doubt there ever will be...[But] we can't take anything away from Ada. It brought with it much of the discipline...The enabling environment is downright beautiful. The strong typing — the inherent and fixed capability of documentation as you go — the ease of maintenance — and integration that won't quit...[and] the technical way of thinking...[Ada] is excellent in its ability to bring together, so smoothly, the efforts of many people and many software developers to get one solution. And most important, it embodies the commitment we've already made [to software engineering].* [EDMONDS93[1]]

In his book Future Shock, Alvin Toffler states, "*Technology feeds on itself. Technology makes more technology possible.*" [TOFFLER70] This is exactly what Ada technology has accomplished in just over a decade. Tools to support the Ada language abound. In a mature Ada software engineering environment, every activity that can be automated should be. A well-managed toolset saves time and increases the quality of your product and the productivity of your team. Napoleon was known for saying,

> *The art of war is a simple art; everything is in the performance. There is nothing vague about it; everything in it is common sense.* [NAPOLEON55]

We use Ada, not for Ada's sake. We use Ada because it performs as an enabling technology to achieve the goals of engineered software and because ***it makes plain common sense!***

REFERENCES

[Ada/C++91] Ada and C++ Business Case Analysis, Deputy Assistant Secretary of the Air Force, (Communications, Computers, and Logistics), Washington, DC, July 1991

[ALLEN92] Allen, Tim, "An Ada Tutorial," *CrossTalk*, August 1992

[ANDERSON94] Anderson, Christine M., information provided by the Ada 9X Project Office, 1994

[AW&ST90] "Lantirn-Equipped F-15Es Post Strong Deterrence to Iraqi Threat," *Aviation Week & Space Technology*, November 12, 1990

[AW&ST94] "F-15 Eagle: For the Long Run," *Aviation Week & Space Technology*, December 12, 1994

[BAKER92] Baker, Emanuel R., "TQM in Mission Critical Software Development," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[BOLEN92] Bolen, Scott, "Ada Software Development: Lessons-learned from the Range Control System Program," white paper, Rome Laboratory/ OCDS, Griffiss AFB, New York, June 1992

[BOOCH94] Booch, Grady and Doug Bryan, Software Engineering with Ada, Third Edition, Benjamin/Cummings Publishing Company, Redwood City, California, 1994

[CAMP89] Camp, Robert C., Benchmarking: The Search for Industry Best Practices that Lead to Superior Performance, ASQC Quality Press, Milwaukee, Wisconsin, 1989

[CONSTANCE95] Constance, Paul, "Survey Confirms Ada is Top DoD Language," *Government Computer News*, May 1, 1995

[DANE92] Dane, Abe, "Strike Eagle," *Popular Mechanics*, July 1992

[DIKEL91] Dikel, Dave, as quoted by Gary H. Anthes, "Ada Making Its Mark at Commercial Sites," *Computerworld*, June 17, 1991

[EDMONDS93[1]] Edmonds, Lt Gen Albert J., as quoted in *Ada Information Clearinghouse Newsletter*, Vol. XI, No. 2, August 1993

[EDMONDS93[2]] Edmonds, Lt Gen Albert J., as quoted by Joyce Endoso, "Ada Gets Credit for F-22's Software Success," *Government Computer News*, April 26, 1993

[ELAM92] Elam, Terence W., and Lt Col Patricia K. Lawlis, "Ada Whips Assembly," *CrossTalk*, March 1992

[GAETANO92] Gaetano, Vivian M., and Cap Carol E. St. Denis, "Ada Portability Problems?" *CrossTalk*, STSC, Number 38, November 1992

[GALORATH95] Galorath, Daniel D., "Ada Versus Fourth Generation Languages" briefing, Galorath Associates Inc., SEER Technologies Division, Los Angeles, California, July 7, 1995

[GAO89] *Submarine Combat System: Technical Challenges Confronting Navy's Seawolf AN/BSY-2 Development*, Report to the Chairman, Subcommittee on Projection Forces and Regional Defense, Committee on Armed Services, US Senate, US General Accounting Office, March 1989

[GROSS92] Gross, Col Richard R., "The Air Force's Ada Policy: Today and Tomorrow," paper presented to Electronic Industries Association Committee Meeting, Hyatt Dulles Airport, Virginia, April 7, 1992

[HOOK95] Hook, Audrey, Bill Brykczynski, Catherine W. McDonald, Sarah H. Nash, and Christine Youngblut, A Survey of Computer Programming Languages Currently Used in the Department of Defense, IDA Paper P-3054, Institute for Defense Analyses, Alexandria, Virginia, January 1995

[HOROWITZ95] Horowitz, Barry M., personal communication to Lloyd K. Mosemann, II, December 1995

[INTERMETRICS95] *Ada 95 Rationale*, Intermetrics, January 1995

[LEONG-HONG93] Leong-Hong, Belkis as quoted by Joyce Endoso, "DISA Targets Data Standards, Ada Renewal," *Government Computer News*, October 25, 1993

[LUDWIG92] Ludwig, Lt Gen Robert H., "The Role of Technology in Modern Warfare," briefing presented to the Software Technology Conference, Salt Lake City, Utah, April 14, 1992

[MAHAN08] Mahan, RADM Alfred Thayer, Administration and War, Little, Brown, and Company, Boston, Massachusetts, 1908

[MOSEMANN89] Mosemann, Lloyd K., II, "Software Engineering and Beyond," *SEI Bridge*, June 1989

[NAPOLEON55] Napoleon I, Christopher J. Herold, editor, The Mind of Napoleon: A Selection from his Written and Spoken Words, Columbia University Press, New York, 1955

[NATO86] *What a Commander Needs to Know About Ada*, SHAPE Technical Center, The Hague, The Netherlands, 1986

[PAIGE93] Paige, Emmett, Jr., "Ada Joint Program Office" memorandum addressed to SAF/AQK, June 29, 1993

[REED89] Reed, Gregg P., "Ada Use Increasing for MIS," Ralph E. Crafts, ed., *Anthology of Commercial Ada Applications: Usage and Issues for Non-Weapons Software Systems*, Software Strategies and Tactics, Inc., Harpers Ferry, West Virginia, 1991

[REIFER95] Reifer, Don, as quoted by Diane Hamblen, "Ada: Dispelling the Myths," *CHIPS*, July 1995

[RIEHLE94] Riehle, Richard, "Ada in Space Systems," *Embedded Systems Programming*, November 1994

[SBM93] Software Business Management, Inc., study results reported in letter to SAF/AQK, February 25, 1993

[SEARS95] Sears, RADM Scott L., "The Ultimate Challenge: Software Engineering and the Navy's BSY-2 Program," speech presented to the 7th Annual Software Technology Conference, Salt Lake City, Utah, April 9, 1995

[SEI90] Carlson, Marthena, and Gordon N. Smith, *Understanding the Adoption of Ada: Results on an Industrial Survey*, SEI Special Report, SEI-90-SR-10, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1990

[TOFFLER70] Toffler, Alvin, Future Shock, Bantam Books, New York, 1970

[WEIDERMAN89[1]] Weiderman, Nelson, *Ada Adoption Handbook, Compiler Evaluation and Selection*, Version 1.0, CMU/SEI 89-TR-13, ESD-TR-89-12, 1989

[YATES91] Yates, Gen Ronald W., keynote address presented to Tri-Ada '91 Conference, San Jose, California, October 22, 1991

Blank page.

CHAPTER 5
Addendum B

Ada Users Throughout the World Tell: WHY Ada?

EDITOR'S NOTE

This article is based on a new video, "Ada — The Language for a Complex World." A 17-minute version was designed for Chief Technical Officers (CTOs) and a 9-minute version for Chief Executive Officers (CEOs). A limited number of copies of the videotape are available through the AdaIC, c/o IIT Research Institute, P.O. Box 1866, Falls Church, VA 22041.

> ***If this monster was written in anything but Ada, I don't know how we'd take care of it.*** — Ken James, General Supervisor, Weirton Steel Hot Mill

The "*monster*" in question is not a giant information system somewhere in the Department of Defense (DoD); it's not a weapons system, or anything that might be covered by the DoD's Ada mandate. It's not in some government agency, or defense contractor. Nonetheless, the demands on this monster are as great as those on many weapons systems, and the consequences of failure will affect people immediately. This *"monster"* is Weirton Steel, a sprawling steel mill in Weirton, West Virginia. It's an employee-owned company, and the economic health of the surrounding valley depends to a large extent on Weirton Steel. Ada was chosen for the software that runs Weirton because it works — because of the reliability of code written in Ada, because of the ease of modifying it to suit changed needs.

Ada CODE IS RELIABLE

> ***...the software keeps running. It's capable of regenerating and going ahead.*** —Bill Zickefoose, Weirton Steel, Process Automation Engineer

It was Ken James, General Supervisor of Weirton's Hot Mill, who referred to the Weirton system as a monster. *"I don't get a lot of phone calls in the night,"* said Bill Zickefoose, process automation engineer. *"Even though we have 24-hour coverage via beepers and things like that, we don't get very many calls. The reason we don't get a lot of calls is not because they're going to sit there with the mill down. It's because the software keeps running. It's capable of regenerating and going ahead."*

Ada CODE IS REUSABLE

> ***For the air traffic control system in Oslo, Norway, Ceselsa has achieved "close to 80% of reuse."*** — Diego Macia, Ceselsa, ATC Systems Manager, International

The need for reliability is even more pressing for air traffic control — highly distributed, real-time systems; but that absolute concern doesn't mean that people in air traffic control ignore economics. In fact, the two work together in the case of reuse. Ceselsa is working on air-traffic control centers all over Spain. *"All the centers are totally interconnected and integrated,"* said Javier Ruano, Ceselsa's ATC System Manager for Spain, and *"the software is 95% in Ada."* Diego Macia, ATC Systems Manager, International, noted that they are also working on the air-traffic control system for Oslo, Norway. There they have achieved *"close to 80% of reuse."* And with other programs, they are reaching an even higher percentage — with programs under development in Amsterdam, Bombay, New Dehli, Hong Kong, and Frankfurt.

The High Quality and Low Errors of Ada Code Give an Edge to the Little Guy

> ***I'd really find it hard to run this company without Ada.*** —Peter Wannheden, co-founder, Paranor Software

One of Ada's virtues is that it makes it easy to integrate the work of large, scattered groups of programmers, working over long periods of time. But Ada works equally well for small companies working with tight deadlines—and gives them a competitive edge in competing with the big guys. Paranor Software had only 15 people and five months to start from scratch and develop a pilot to compete against two large companies for a program of the post, telephone, and telegraph (PTT) administration in Switzerland. The PTT acts as a banking system, as well as a phone and postal service, and it needed to computerize its financial records for a half-million customers.

Said Paranor's co-founder, Peter Wannheden, *"One contributing factor may be that we convinced the customer that our software methodology — including the use of Ada — would guarantee a high-quality product."* And that has turned out to be true. *"We've had a very, very low rate of errors which were determined after installation of the software,"* said Mr. Wannheden. *"I'd really find it hard to run this company without Ada. I don't know what we would have here."*

QUALITY COMPILERS ARE AVAILABLE

> ***It had compilers of the quality which we absolutely must have to develop safe and essential systems.*** — Vin Joag, Smiths Industries, FMV Software Manager

Ada's strengths are not merely in the software-engineering principles it supports, but that it's *a standard.* Users don't buy merely a language; they buy and use *compilers.* The process of validating compilers contributes to Ada's usefulness on the job. For Vin Joag, FMV Software Manager for Smiths Industries, this was *"one of the major reasons we selected Ada ... we found that the language is very defined. It had compilers of the quality which we absolutely must have to develop safe and essential systems."*

Ada CODE IS PORTABLE

> ***The Ada has been written just once, and we're then using three different compilers.*** — Noel Wright, GEC Marconi Avionics, Engineering Manager

The Ada standard has also encouraged portability. When we think of portability, most of us probably think of using code on one platform successfully and then later porting it to another. At least one company, however, found themselves using the same code in the same program on three different processors.

The flight-control system for the Boeing 777 uses three different processors for redundancy. *"The Ada has been written just once,"* said Noel Wright, Engineering Manager at GEC Marconi Avionics, *"and we're then using three different compilers. One of the big things of Ada was to make sure that you could actually port it around very easily."*

Ada SUPPORTS TASKING

> ***Our application has a few hundred concurrent threads. ...there just isn't any good way to manage that, except Ada.*** — John Malcolmsen, JEOL, Assistant Manager/NMR Software R&D

Going from the skies down to the microscopic world, you'll find Ada doing jobs that can't be done otherwise. This was the case for JEOL, the fifth-largest analytical-instrument company in the world. In one case, they're looking for the structure of protein molecules. This information is used in studying diabetes and other diseases. *"With Ada, we were able to build software that became intuitively obvious,"* said Bill Bearden, operations manager. John Malcolmsen, assistant manager/NMR software R&D, pointed out that, *"We needed to have tasking, and the only we way we were going to get tasking was to go with Ada. Some applications may have a few dozen concurrent threads; our application has a few hundred concurrent threads. And with that level of concurrency, there just isn't any good way to manage that, except Ada."*

Ada TASKING CAN BE FUN, TOO

> ***Ada has been really the reason that you can make Paintball here.*** — Dave McAllister, Silicon Graphics, Inc., Visual Magic Division, Enabling Technologies Program Manager

But neither Ada nor tasking has to be all work and no play. In fact, it can be a game - in this case, at Silicon Graphics, Inc. (SGI). Dave McAllister, Enabling Technologies PM at SGI's Visual Magic Division, explained Paintball: *"The basic premise of this was a multi-player, multi-tasking, virtual reality game. We actually presented it to the rest of the engineering team here at SGI, and we were told ... that it was a great concept, but it would never work."* That might have been a sound judgment — except for Ada. *"Ada has been really the reason that you can make Paintball here,"* according to Mr. McAllister. *"Ada itself is the first language designed for multiple processing, and we're actually running on a system with four CPUs in it."* He pointed out that there are *"anywhere from 200 to a thousand Ada tasks appearing, doing something inside the graphics."* And just as reuse worked with air-traffic control, it worked with games — and with a game quite different from Paintball. Fireflight is a *"very different problem, a mission-planning scenario,"* said Mr. McAllister. But Fireflight *"reused roughly 65% of the code we wrote for Paintball."*

AND NOW WE GET THE BENEFITS OF Ada 95

With the advent of Ada 95, additional benefits are on the way. Bill Carlson, vice president/CTO of Intermetrics, Inc., said that, *"the main thing that has changed"* since 1983 is the ready availability of compilers. *"Today, anybody can get a good Ada compiler, and in fact, you can get one for free"* — referring to the Ada 95 GNAT compiler available on the Internet.

Tucker Taft, Chief Language Designer for Ada 9X (Ada 95), pointed out that Ada 95 builds on Ada 83's excellence in handling *"a large system where you had relatively well understood requirements and a very disciplined engineering process."* Ada 95 continues that, *"but it's also oriented towards, let's say, the smaller system. The more dynamic system, the system where the requirements aren't as well defined in advance. I don't think you should think of Ada [95] as a new language. It's really Ada 83 with some restrictions removed, some new features added, and basically sort of a more open feel."* Author John Barnes, in updating one of his books on Ada, offers that the *"delightful"* point is *that "old difficulties have been cast aside"* and he found himself on *"much friendlier ground."*

Ada 95 — Building on Strength

Many of the strengths of Ada 95 are based on the strengths of Ada 83 — one of which was strong typing. According to Stuart Liroff, Engineering Manager at Silicon Graphics:

> *"One of the reasons Ada 95 is so powerful is due to the strong typing, and the inheritance that it offers you from derived types."* He noted that, *"the reason this is so important is because typing allows you to characterize the objects that you're modeling in your environment or your application. And Ada 95 has strong typing — much stronger typing than any of the other languages."* The result, said Mr. Liroff, is that *"you'll find that with strong typing, you will get more correct and more true code out of your engineers."*

CHAPTER 5
Addendum C

The Ada 95 Philosophy

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 5
Addendum D

Ada Implementation Lessons-Learned from SSC and CSC

S. Tucker Taft

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

Blank page.

CHAPTER 6

Risk Management

CHAPTER OVERVIEW

Software acquisition and management may be the greatest challenge of your career. Software programs are more prone to failure than to success. Therefore, the most important thing you have to manage in your program are the ***inherent risks*** *associated with the development and support of all major software-intensive systems. A risk is the probability of an undesirable event occurring and the impact on the success of your program if that risk occurs. Your program is subject to three main risk factors: (1) your product does not meet planned performance requirements, (2) your program overruns its budget, and/or (3) your product is delivered too late or not at all.*

In this chapter you will learn that risk management encompasses a defined set of activities which include identifying, analyzing, planning, tracking, controlling, and communicating program risks. Risk mitigation techniques include avoidance, control, assumption, and transference. How you implement these activities and techniques are defined in structured, disciplined risk management methodologies. The risk management methodology you choose must be tailored and selected based on those risks peculiar to your program. The sampling of methodologies discussed here are systematic, repeatable, and based on solid, proven risk management techniques. A disciplined risk management approach is essential for program insight so you can make decisions and take preventive actions critical to program success.

All the methods presented here stress that significant risks be assessed and their impacts quantified in terms of quality, cost, and schedule. Risk management must be planned, budgeted, and accommodate abatement plans based on quantified risk impacts. Time and dollar requirements for risk management must be included in the total estimated contract cost. Risk elements must also be tracked throughout the total system life cycle. Techniques for managing risk are discussed in subsequent chapters and throughout these Guidelines. They include: software development maturity assessments; engineering discipline; process, product, and program monitoring through measurement and metrics; reuse; strategic and continuous program planning; reviews, audits, and peer inspections; defect prevention, detection, and removal; and constant process improvement.

CHAPTER 6 Risk Management

Risk management, on an iterative tradeoff basis, is key. It is a proactive way for you to prevent problems and be prepared. It is an insurance policy. It is a way for you to be armed with alternative solutions if an anticipated (or unforeseen) problem interrupts or negates your plans. The time, effort, and funds you dedicate to its practice will be well-spent. ***Risk management is a necessary, sound investment in your program's success.***

CHAPTER 6

Risk Management

RISK MANAGEMENT: An Investment in Success

It was one of those sweltering, breezeless, smog-ladened New York City summer afternoons when no one wanted to be outdoors. Those who are out staggered from air conditioned offices to air conditioned taxis heading towards refrigerated apartments. One man was sitting out in that heat. He was slumped wearily, glassy-eyed, his brief case in his lap, his power tie askew, his Amante suit jacket thrown over the back of a bench. He looked like someone who had been squeezed through the slot of an automatic teller machine. He was oblivious to the heat or anyone around him. He had not slept in two days and emitted a deep sense of despair. This was a severely troubled man. He shook his head from side to side repeating, *"If I only knew how it happened ... if I only knew how it happened..."*

Our man was one of the senior executives at Bank of America who had just lost his high-paying job because of a software acquisition disaster. Other nonexecutive staff members would be receiving pink slips in the morning. The bank's chairman was not amused when our executive had to make the hard decision to abandon an originally estimated $20 million software-intensive system development after spending $60 million trying to make it work. [LATIMES88] This decision followed an earlier failed attempt that cost the bank an additional $6 million. He really had been backed into a corner. The second attempt was originally scheduled to be a two-year effort, but they were three years into the resurrected development—with no end insight. The chairman had very publicly declared that Bank of America was going to be the banking industry's information technology leader and this software-intensive system was the main strategic move towards getting them there. Having to admit this monumental

investment loss through their inability to field the system was a blow to the bank's integrity, requiring painful explanation to investors.

As you learned in Chapter 1, *Software Acquisition Overview*, and in the *Scientific American* article, "Software's Chronic Crisis," in the Foreword, the Bank of America fiasco is not an isolated case. Software development is one of the most (if not ***THE*** most) risk prone management challenge of this decade. Risk factors are always present that can negatively impact the development process, and if neglected, can tumble you unwittingly into program failure. Software developments, like campaigns and battles, are nothing but a long series of difficulties to overcome. To counteract these forces, you must actively assess, control, and reduce software risk on a routine basis. As Chief of Staff of the Army, often called "*The Organizer of Victory,*" General George C. Marshall, explained to the first officer candidate class at Fort Benning, Georgia,

> *Campaigns and battles are nothing but a long series of difficulties to be overcome. The lack of equipment, the lack of food, the lack of this or that are only excuses; the real leader displays his quality in his triumphs over adversity, however great it may be.* [MARSHALL41]

Also in Chapter 1, you were told why software programs fail. **Poor management** is cited time and again as the culprit causing software acquisitions to go belly up. Our development processes are immature, chaotic, and unpredictable. Our estimates of cost, schedule, and software size and complexity are inadequate. Our problem solving and decision making is poor because we do not plan, measure, track, or control the process and product. Risk management addresses all these shortcomings. To be an effective manager, you must identify and mitigate risks throughout the entire life cycle. You will be hard pressed to eliminate all risks, but should take action on those risks most critical to the success of your program to the point where they become manageable.

Implementing **risk management** in your program (based on a judicious mix of theoretical background, practical methodologies, and ***common sense***) will give you a greater chance to succeed with the ability to straighten things out. It is important to form an effective government/industry team at the onset of the acquisition. Sharing information and concerns, careful listening, and timely responses between mutually bound partners are essential risk management activities. Industry's belief that their concerns about risk will be

addressed by the Government is also vital. Conversely, the SPO's ability to rely on its industry partner(s) to provide quality solutions (within the parameters provided by the user) enhances the probability of risk management success. Figure 6-1 shows how effective risk management can minimize the cost of risk to a program.

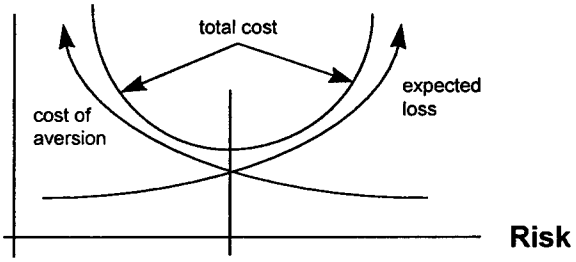


Figure 6-1 Benefits of Effective Risk Management [HALL95]

SOFTWARE RISK

While risk management has been used for years in many different disciplines and professions (including financial, petrochemical, insurance, etc.), and has been advocated by DSMC for over 12 years, it has only been in the past 5 to 7 years that risk management is receiving more attention in the software engineering community. [MARCINIAK94] ***Risk is defined as the probability of an undesirable event occurring and the impact of that event occurring***. A risks is the precursor to a problem. It is the probability that, at any given point in the system life cycle, its predicted goals (either operational or logistical) cannot be achieved within available resources. There is a high probability that you will have less than a full understanding of the requirements of either the software product or the process before you begin your development. You also run the risk that it will take longer and cost more than expected. Trying to totally eliminate risk is a futile endeavor — however, managing risk is something you can and must do. To know whether an event is truly *"risky,"* you must have an understanding of the potential consequences of the occurrence/nonoccurrence of that event. As a program manager, you risk failure in three ways and combinations thereof:

- The product does not meet performance requirements (operationally or logistically),
- Actual costs are higher than budgeted, and
- Delivery of the product is too late.

Software Risk Factors

Software risk factors that impact a product's performance, cost, and schedule can be further segmented into five risk areas. However, any given risk may have an impact in more than one area. The five risk areas are:

- **Technical risk** (performance related),
- **Supportability risk** (performance related),
- **Programmatic risk** (environment related),
- **Cost risk**, and
- **Schedule risk**.

The following sections identify: the common risk factors found in many programs; a "*Top-10*" risk identification checklist; environmental factors that sometimes get overlooked; and additional interrelated factors that contribute to software risk. While some of these risk factors may overlap, they all must be considered.

Common Risk Factors

The **Software Program Manager's Network** (SPMN) identifies common threads among troubled software programs. These conclusions, based on risk assessments performed on 30 software programs since 1988, include:

- **Management**. Management is inconsistent, inappropriately applied or not applied at all. In many cases, it is reactionary — management reacts to rather than plans for issues.
- **Predictable risks ignored**. When a problem arises, it is identified by program personnel; but they often say, "*Hey, it's too much trouble. I can't deal with it.*" They ignore it, press on, and then get blind-sided by the impact.
- **Disciplines not uniformly applied**. On many programs, configuration management, product assurance, and technical disciplines are not uniformly applied. Organizations throw away standards as they go through the program to "*buy*" more schedule, performance, or save on cost. The result is the program moves

along in the short-term, but creates a rolling wave of disaster in the long-term.

- **Poor training**. In many cases, the reason managers do not perform a specific task is because they did not know how to do it. For example, some managers do not understand costing or schedule, do not know how to do the technical job, or to plan. No accessible training or resources are available to them.
- **Fallacy of easy solutions**. ***Software programs often get in trouble when generic solutions are applied to specific problems.*** Methods designed for non real-time work are used in real-time programs. Unproven, Silver Bullet techniques with no tool support, standards, or an experience base from which to proceed are used on high risk programs. In addition, programs having difficulty often fail to scale the work to resources.
- **Inadequate work plans**. Critical constraints and work plans include schedules, budgets, work allocation, and limited, time-sensitive resources. Programs in trouble do not have a clue where they stand. Inadequate schedules, or schedules not enforced, are often the problem.
- **Schedule reality**. The schedule plan must be realistic ,and if the schedule slips, the impact on delivery must be assessed. Programs in trouble do not deal with this. If there is a schedule slip, they do not realistically consider the effect on the end-date. They take short cuts and came up with *success-oriented* schedules to avoid announcing an end-date slip.
- **Delivery focus**. With many programs in trouble, focus on the schedule and process, not on the delivery. Successful programs focus on the incremental completion of an event. The relationship between all activities should be that one completion, which feeds the next completion, and the next.
- **Constraints**. Successful programs use reasonable metrics to status and analyze the program, assess product quality and process effectiveness, and to project the potential for success. Programs in trouble abuse metrics which are used to justify unreasonable positions. The bad is ignored and the good is inflated to overshadow the bad.
- **Customer responsibility**. The customer should not just sit back and oversee. The customer has to provide the hierarchy of documentation and at least provide standards for the software development phase in which they are interested. In almost every troubled program, the customer has a "*hands-off*" approach.
- **Methods and tool selection**. Almost every troubled program has problems in this area. The tools selected are inappropriate for

the job. The program staff has too little or no experience with the methods used, which are not integrated, but run as stovepipes. The process is not integrated through configuration management — no effective information flow within the program is established. [EVANS94]

"Top-10" Risk Identification Checklist

Boehm identifies a "*Top-10*" list of major software development areas where risk must be addressed. [BOEHM91]

- Personnel shortfalls,
- Unrealistic schedules and budgets,
- Developing the wrong software functions,
- Developing the wrong user interface,
- Goldplating,
- Continuing stream of requirement changes,
- Shortfalls in externally furnished components,
- Shortfalls in externally performed tasks,
- Real-time performance shortfalls, and
- Straining computer science capabilities.

Environmental Factors

Charrette explains there are subtle environmental factors often overlooked when identifying sources of risk. They include:

- **Software developments are very complex**. The software problem has numerous elements with extremely complicated interrelationships.
- **Problem element relationships can be multidimensional**. The changes in elements are not governed by the laws of proportionality. It is well documented that adding more people to a program that is behind schedule, in many instances, will make it even later.
- **Software problem elements are unstable and changeable**. Although cost and schedule may be fixed, actual costs in labor and time to complete are difficult to program.
- **The development process is dynamic**. Conditions ceaselessly change; thus, program equilibrium is seldom achieved. The environment is never static — hardware malfunctions, personnel quit, and contractors do not deliver.
- **People are an essential software development element and a major source of risk**. Economic or technical problems

are easy with which to deal. The higher-level complications, multidimensional ambiguities, and changing environment caused by conflicting human requirements, interaction, and desires are what cause problems. Software development is full of problems because it is a very ***human endeavor***. [CHARETTE89]

Interrelated Factors

Additionally, there are other **interrelated factors** that contribute to software risk. These factors are:

- **Communication** about risk is one of the most difficult, yet important, practices you must establish in your program. People do not naturally want to talk about potential problems. Rather than confronting imaginary problems while they are still in the risk stage, they wind up having to deal with them after they become full-blown, real problems. Then there is a lot of communication! Effective risk planning only occurs when people are willing to talk about risks in a non-threatening, constructive environment.
- **Software size** can affect the accuracy and efficacy of estimates. Interdependence among software elements increases exponentially as size increases. With extremely large software systems, handling complexity through decomposition becomes increasingly difficult because even decomposed elements may be unmanageable.
- **Software architecture** also affects software risk. Architectural structure is the ease with which functions can be modularized and the hierarchical nature of information to be processed. It is also development team structure, its relationship with the user and to one another, and the ease with which the human structure can develop the software architecture. [PRESSMAN93]

MANAGING SOFTWARE RISK

While you can never totally remove software risk, there are several different techniques that can be used to mitigate it. These techniques, of course, should be used in the structure of a software risk management process. This section identifies risk mitigation techniques and the basic software risk management process. While many methods exist, the next section, *Formal Risk Management Methods*, presents several structured, well-proven methods from which you can choose for your program.

Risk Mitigation Techniques

Risk mitigation techniques include:

Risk avoidance. You can ***avoid the risk*** of one alternative approach by choosing another with lower risk. This conscious choice avoids the potentially higher risk; however, it really results in risk reduction in risk — not complete risk elimination. While a conscious decision to ignore (or assume) a high risk may be a creditable option, an unconscious decision to avoid risk is not. As General Robert E. Lee proclaimed,

> *There is always hazard in military movements, but we must decide between the possible loss of inaction and the risk of action.* [LEE33]

You must assess, rate, and decide on the possible consequences of inaction. You must also decide if the benefits of acting on a risk merit the expense in time and money expended. You and your developer should document all risk handling actions with supporting rationale. You should also employ risk management in concert with metrics and process improvement used to measure, track, and improve your program's progress and process.

Risk control. You can ***control risk*** (the most common form of risk handling) by continually monitoring and correcting risky conditions. This involves the use of reviews, inspections, risk reduction milestones, development of fallback positions, and similar management techniques. Controlling risk involves developing a risk reduction plan, then tracking to that plan.

Risk assumption. You can ***assume risk*** by making a conscious decision to accept the consequences should the event occur. As Napoleon explained,

> *If the art of war consisted merely in not taking risks, glory would be at the mercy of very mediocre talent.* [NAPOLEON55]

Some amount of risk assumption always occurs in software acquisition programs. It is up to you to determine the appropriate level of risk that can be assumed in each situation as it presents itself.

Risk transference. You can ***transfer risk*** when there is an opportunity to reduce risk by sharing it. This concept is frequently used with contractors where, for instance, contract type, performance incentives (including award fees), and/or warranties are risk sharing contractual mechanisms. Although many of these techniques only share *cost risk*, risk transfer is often beneficial to the Government and the developer.

Risk Management Process

It is your responsibility to put a process in place that enables you and your team to identify, analyze, plan, track, and relentlessly control risk. Risk management costs time and money. However, *it is always less expensive to be aware of and deal with risks than to respond to unexpected problems*. A risk that has been analyzed and resolved ahead of time is much easier to deal with than one that surfaces unexpectedly. [BLUM92] Figure 6-2 identifies a basic risk management model. This model identifies the fundamental risk management actions you must take:

Figure 6-2 Risk Management Continuous Process

- **Identify**. Search for and locate risks before they become problems adversely affecting your program.
- **Analyze**. Process risk data into decision-making information.
- **Plan**. Translate risk information into decisions and actions (both present and future) and implement those actions.
- **Track**. Monitor the risk indicators and actions taken against risks.
- **Control**. Correct for deviations from planned risk actions.
- **Communicate**. Provide visibility and feedback data internal and external to your program on current and emerging risk activities. *[Refer to CMU/SEI-93-TR-06.]*

Another, more formal risk management model is shown in Figure 6-3. This model illustrates more of the functions of, and interrelationships among, the basic steps. This model is representative of many formal risk management methods *[described in the following section]*.

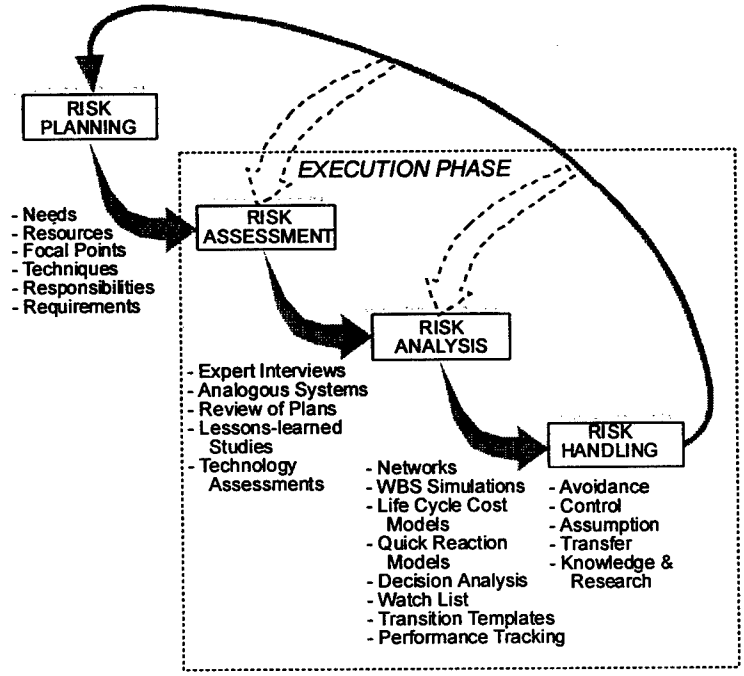


Figure 6-3 A Formal Risk Management Process

FORMAL RISK MANAGEMENT METHODS

A proactive approach to risk is the most effective way to control it. The methods you employ must be disciplined, systematic, repeatable, and based on solid principles. They must facilitate communication among all stakeholders—at all levels. A disciplined risk management approach will help you to obtain valuable program insight, allowing you to make decisions and take actions that may be critical to its success. As Sir Winston Churchill explained,

> *...when fortune is dubious or adverse; when retreats as well as advances are necessary; when supplies fail, arrangements miscarry, and disasters impend... —as the*

severity of military operations increases, so also must the sternness of the discipline. [CHURCHILL99]

Formal risk management methods provide the necessary information to focus on priority risks and their mitigation. They define distinct procedures to aid in performing risk management functions and have integrated tools and techniques to ensure standardized application. Formal methods gain their robustness through practical use, testing, and continuous field validation. As they evolve and mature, they demonstrate their ability to produce similar, consistent results no matter who applies them. There are many risk management methods. Keep in mind that whichever method you choose, it must be tailored to your specific program needs. Examples of these methods include:

- Software Risk Evaluation (SRE) method,
  - Example: Harris Corporation risk management streamlining,
- Boehm's Software Risk Management method,
- Best Practices Initiative Risk Management method,
- Team Risk Management method,
  - Example: Team Risk Management on the NALCOMIS Program, and
- B-1B Computer Upgrade Risk Management method and example.

Software Risk Evaluation (SRE) Method

The SEI has developed a **Software Risk Evaluation (SRE)** method which focuses on the contractor/government relationship. It is a formal approach for identifying, analyzing, communicating, and mitigating the technical risks associated with a software-intensive acquisition. As illustrated in Figure 6-4 (below), the program manager (Government) directs an independent SRE team to perform a risk evaluation. The team then executes SRE functions for the contractor's target software development task. The outcome is a set of findings processed to provide value-added information (results) to the Government. The SRE can also be used as a business tool by the contractor to manage software program risks. *[Refer to CMU/SEI-94-SREv0.2, Software Risk Evaluation Method.]*

The SRE method is based on the Risk Management Paradigm *[discussed below],* the Software Development Risk Taxonomy, and the Taxonomy-Based Questionnaire. The later two elements are defined as:

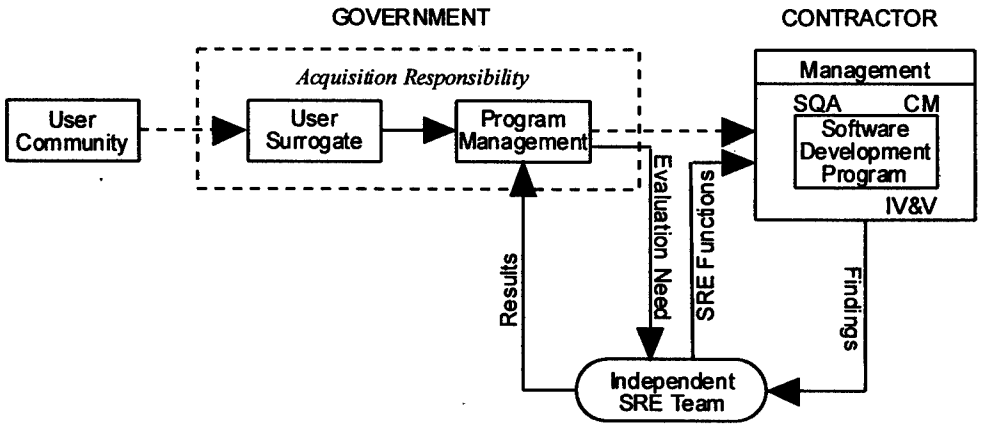


Figure 6-4 SRE Method Application

- **Software Development Risk Taxonomy (SDRT).** The taxonomy provides a basis for organizing and studying the breadth of software development issues. As illustrated in Figure 6-5, it serves as a systematic way of eliciting and organizing risks and provides a consistent framework risk management method and technique development. *[See Table 6-7 for a more detailed, lower-level taxonomy.]*

- The SEI **Taxonomy-Based Questionnaire (TBQ).** The TBQ is a tool specifically used for identifying software development risks. This tool ensures all potential risk areas are covered by asking questions at the SEI Software Development Risk Taxonomy detailed attribute level. The TBQ also contains specific cues and follow-up questions that allow the person administering the questionnaire to probe for risks. This tool is effective when used along with appropriate techniques for interviewing management and technical program personnel. Field tests have shown the TBQ produces better results when administered by an independent team and the respondents are in peer group sessions. *[For more information, contact the SEI (see Volume 2, Appendix A) or look them up online (see Volume 2, Appendix B).]*

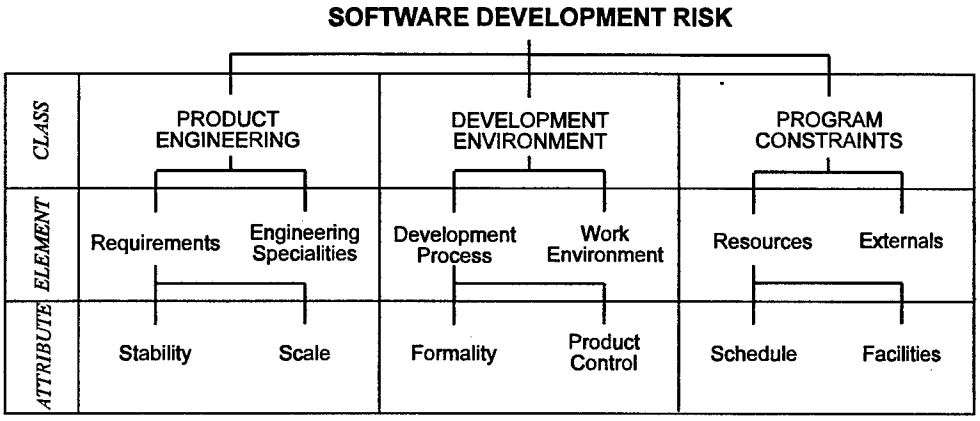


Figure 6-5 Structure of the Software Risks Taxonomy

SRE Functional Components

The SRE method consists of primary and support functions, as illustrated on Figure 6-6. **Primary SRE functions** are:

- **Detection** is the function of finding target program software technical risks. This function ensures systematic and complete coverage of all potential technical risks areas. It also ensures efficiency and effectiveness through use of appropriate tools and techniques. Risk detection in the SRE method is performed by using the following:
  - SEI **Taxonomy-Based Questionnaire** to ensure complete coverage of all areas of potential software technical risks;
  - An **SRE team** that conducts group interviews using a specified technique and implementation process which includes functional roles for each individual; and
  - Selection of appropriate **individuals** and **guidelines** for interview groups to ensure coverage of all viewpoints including software development, support functions, and technicians and managers.
- **Specification** is the function of recording all aspects of identified technical risks including their condition, consequences, and source. Each risk is assigned to a specific category within the SEI Software Development Risk Taxonomy, i.e., the source of the risk is specified as belonging to a taxonomy class, element, or attribute.
- **Assessment** is the function that determines the magnitude of each software technical risk. By definition, magnitude is the product of severity of impact and the probability of risk occurrence where:

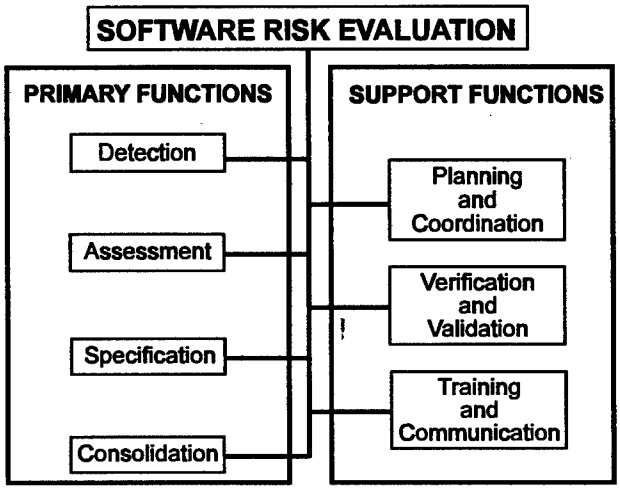


Figure 6-6 SRE Functional Components

Magnitude = Severity of Impact • Probability of Occurrence

Risk statements are assessed at one of three levels of magnitude: high, medium, or low. The level at which a particular risk is assessed depends on the separate assessments of its severity of impact and its probability of occurrence. ***Severity of impact*** is the effect of the particular risk on the target program or task and is assessed on the basis of its impact on software performance, support, cost, and schedule. Each risk is assessed to belong to one severity level: catastrophic/critical, marginal, or negligible. ***Probability of occurrence*** is the certainty or likelihood of the risk becoming true. Each risk is assigned to one probability level: very likely, probable, or improbable

Figure 6-7 illustrates SRE risk magnitude levels. The shaded areas indicate different levels of magnitude. The figure also shows the derivation of the magnitude levels using their assessments for severity of impact and probability of occurrence. For example, a high magnitude risk is one whose severity is catastrophic/critical and probability is very likely or probable. Similarly, a risk is of high magnitude if its severity is marginal and probability is very likely.

- **Consolidation** is the function of merging, combining, and abstracting risk data into concise decision-making information. This is necessary due to the multiplicity of risk detection activities

PROBABILITY / SEVERITY	Very Likely	Probable	Improbable
Catastrophic/ Critical	HIGH		
Marginal		MEDIUM	
Negligible			LOW

Figure 6-7 Risk Magnitude Level Matrix

which cause related risks to be identified from different sources. For example, similar risks are often identified during different interview sessions.

SRE support functions include:

- **Planning and coordination** is preparing for the SRE by selecting the team, scheduling individuals for interview sessions, and arranging for site visits. Two functional roles perform planning and coordinating functions:
  - The **SRE Team Leader** is an experienced person from an independent organization responsible for overall planning and coordination of SRE implementation activities.
  - The **Site Coordinator,** from the target organization, is the single interface with the SRE Team Leader who performs planning and coordination activities including scheduling individuals for interviews and arranging the site visit.
- **Verification** and **validation** ensures implementation process quality and result accuracy and validity. It provides decision-makers with reliable information for mitigating technical risks. Included in this function are the tools and techniques use to take corrective measures during early stage implementation. For example, techniques, such as risk playback and team reviews, ensure content, structure, attributes, and risk context.
- **Training** and **communication** ensures implementation process effectiveness by making sure all personnel have sufficient knowledge, understanding, and skills. It creates an environment for effective information dialogue and exchange necessary for SRE implementation. Included in this function are software risk evaluation team training, management and technical personnel orientation, and client organization briefings. Also included are the

specific activities, tools, and techniques used to ensure proper training and communication. For example, techniques such as an SRE overview script which the interviewer reads or paraphrases is used at the beginning of an interview session. This creates a proper risk detection environment where respondents understand the interview process and are able to openly discuss software technical risks.

Harris Corporation SRE Risk Management Streamlining Example

Risk management can save money and help build better software with a relatively low investment. Incorporating disciplined engineering risk analysis and management techniques into your management process can earn you 50% or more in **productivity gains**, and greatly increase your potential for producing a quality product. [CHARETTE89] Without effective risk management, Norman Augustine's ***Law of Counter-productivity*** takes over, where "*It costs a lot to build bad products.*" [AUGUSTINE83]

Hall and Ulrich explain how, in 1992, the Harris Corporation's, Information Systems Division, collaborated with the SEI to improve their risk management methods. One lesson-learned from this effort was that risk identification and assessment should be more continuous than had been originally recommended. Factors contributing to a more routine risk management effort were changes and growth in Harris' staffing, an increased awareness of program risk issues, and a different life cycle focus. To increase cost-effectiveness, Harris tailored the SEI **Software Development Risk Taxonomy** identification process illustrated on Figure 6-8. After tailoring, they identified a similar number of risks while using less assessment team members. Savings in manpower were possible because they had a well-defined, documented, and trained risk assessment process. Thus, less time was needed for process explanation and observation. Independence of the assessment team (from the program teams) was another important factor in achieving cooperation and objectiveness in risk identification. Repeated **Software Engineering Process Group (SEPG)** leadership of successive risk assessments assured consistency in implementation and complemented process improvement. Because, ***risk management is not a one size fits all***, Harris also tailored their risk management program to fit both large and small software development programs. It was customized to fit the dynamics of integrated product teams and accommodate subcontractor organizations.

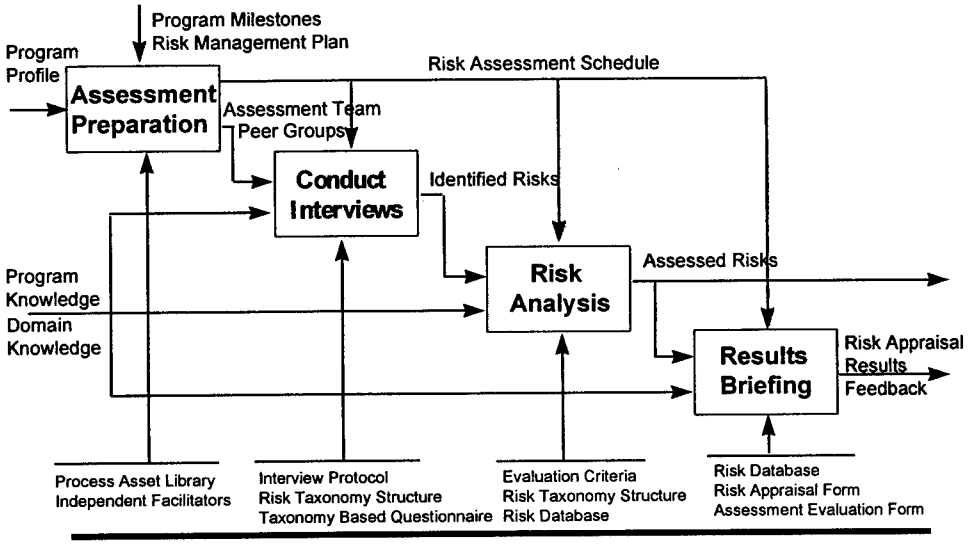


Figure 6-8 Streamlined SEI Risk Assessment Process

Harris also assigned an independent **Risk Champion** who provided an objective and motivational view of the program without placing attribution or blame on individuals whose areas of the responsibility were inherently risky. Motivation was provided to the program in two ways. First, the Risk Champion led discussions to identify or analyze risk situations. Second, the periodic appearance of the Risk Champion served as the catalyst to perform risk management on a routine basis. As one Risk Champion explained, *"I have to drag my people kicking and screaming into a risk meeting, but they're always glad I did."* [HALL95]

Collaboration is a proactive and synergistic way to produce cost-effective results by sharing resources. The SEI/Harris technical collaboration proved to be beneficial for both organizations in understanding effective risk management methods. At Harris, streamlining the risk assessment process reduced the cost of adopting the standard SEI risk identification process. Cost savings were 50% on the assessment team and 25% on interview session time while identifying the same number and types of risks. [HALL95]

Boehm's Software Risk Management Method

Barry Boehm's method of risk management embodies the fundamental concept of **risk exposure**. [BOEHM91] Risk exposure (RE) is defined by the relationship

$$RE = P(UO) \cdot L(UO)$$

where P(UO) is the probability of an unsatisfactory outcome and L(UO) is the loss to stakeholders affected by an unsatisfactory outcome. An ***unsatisfactory outcome*** differs for various classes of participants:

- **Customers** and **developers** see budget overruns and schedule slips as unsatisfactory;
- **Users** see the wrong functionality, user-interface problems, performance shortfalls, or poor reliability as unsatisfactory; and
- **Maintainers** see poor quality as unsatisfactory.

Risk Management Paradigm

Boehm's risk management paradigm is the decision tree. He uses the controlling software for a satellite experiment as an example of a potentially risky element. Because the software team is inexperienced, the satellite program manager estimates there is a probability P(UO) of 0.4 (on scale of 0.01 to 1.0) that the software will have a critical defect(s) (CD). He feels this defect will be so critical that it could shut the entire experiment down causing an associated loss L(UO) of the total $20 million program. As illustrated on Figure 6-9, the manager identified two major options for reducing the risk of losing the experiment:

- **Practice better development methods.** This incurs no additional cost and he estimates it will reduce the defect probability P(UO) to 0.1.
- **Hire an IV&V contractor to find and remove latent defects.** This will cost an additional $500,000. Based on the results of similar IV&V efforts, this will reduce the P(UO) to 0.04.

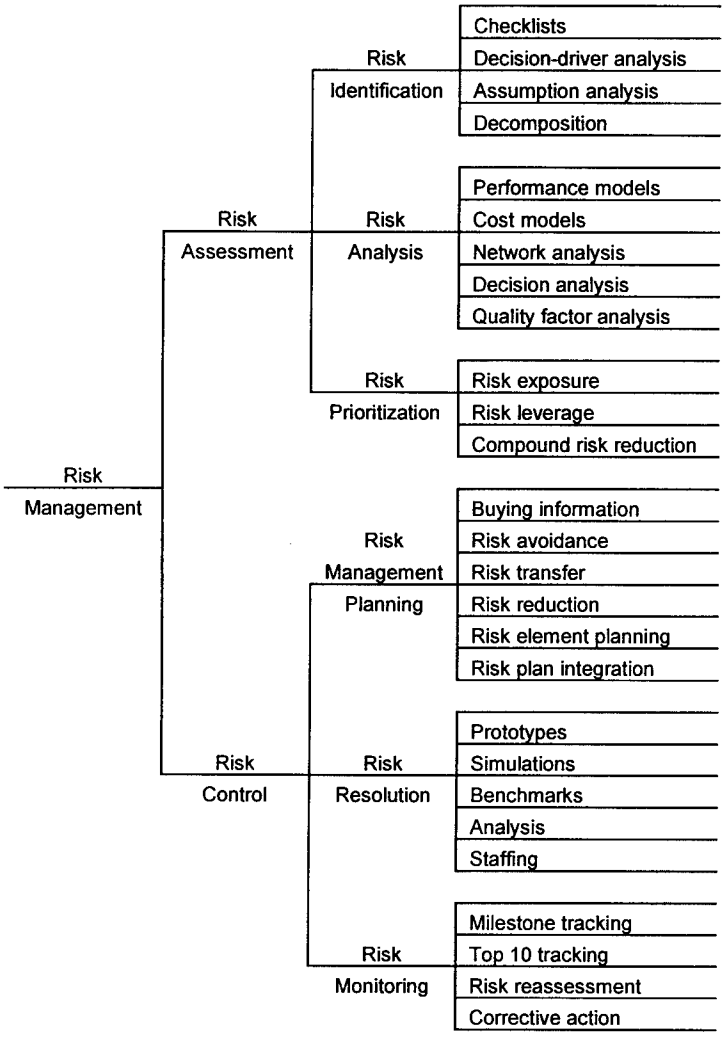


Figure 6-9 Decision Tree for Satellite Software Risk Item

Boehm's Risk Management Process

Boehm's risk management process involves two primary steps, each with three subsidiary steps as illustrated in Figure 6-10 (below). The first primary step, risk assessment, involves risk identification, risk analysis, and risk prioritization. The second primary step involves risk management planning, risk resolution, and risk monitoring.

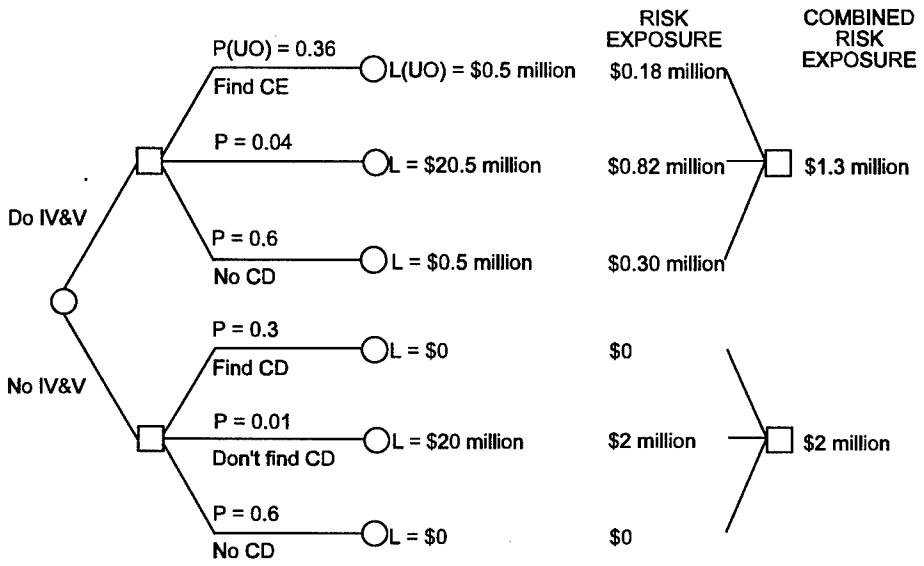


Figure 6-10 Software Risk Management Steps

Table 6-1 illustrates Boehm's **Top-Ten Risk Identification Checklist**. The checklist can be used to help identify the top-ten risk items that could endanger your program's quality, cost, and schedule goals — independent of your contractor's Risk Management Plan. It also provides a set of corresponding risk management techniques proven to be most successful in avoiding or resolving each particular source of risk. Using the checklist you can rate your program's status for the individual attributes associated with its requirements, personnel, reusable software, tools, and support environment.

Table 6-2 (below) illustrates the risk rating and prioritization process for the satellite control experiment. It summarizes several unsatisfactory outcomes and their corresponding ratings for P(UO) and L(UO) (on a scale of 1 to 10) and their resulting risk exposure estimates. Several key factors in risk analysis are illustrated here. One of the highest P(UO)s is data reduction defects, but because these defects are recoverable and not mission-critical, the L(UO) is low with a resulting low RE. Conversely, a low-profile item like a poor user interface becomes a high priority risk item because its combination with a moderately high P(UO) and a medium L(OU) yields a RE of 30.

RISK ITEM	RISK MANAGEMENT TECHNIQUES
Personnel shortfalls	Staffing with top talent; job matching; team building; morale building; cross-training; pre-scheduling key people
Unrealistic schedules and budgets	Detailed, multisource cost and schedule estimation; design-to-cost; incremental development; software reuse; requirements scrubbing
Developing the wrong software functions	Organizational analysis; mission analysis; operations concept formulation; user surveys; prototyping; early user's manuals
Developing the wrong user interface	Task analysis; prototyping; scenarios; user characterization (functionality, style, workload)
Goldplating	Requirements scrubbing; prototyping; cost/benefit analysis; design-to-cost
Continuing stream of requirement changes	High change threshold; information hiding; incremental development (defer changes to later increments)
Shortfalls in externally furnished components	Benchmarking; inspections; reference checking; compatibility analysis
Shortfalls in externally performed tasks	Reference checking; pre-award audits; award-fee contracts; competitive design or prototyping; team building
Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping; instrumentation; tuning
Straining computer science capabilities	Technical analysis; cost-benefit analysis; prototyping; reference checking

Table 6-1 Top-Ten Risk Identification Checklist

UNSATISFACTORY OUTCOME (UO)	P(UO)	L(UO)	Risk Exposure
Software defect kills experiment	3-5	10	30-50
Software defect loses key data	3-5	8	24-40
Fault-tolerant features cause unacceptable performance	4-8	7	28-56
Monitoring software reports unsafe condition as safe	5	9	45
Monitoring software reports safe condition as unsafe	5	3	15
Hardware delay causes schedule overrun	6	4	24
Data reduction software defects cause extra work	8	1	8
Poor user interface causes inefficient operation	6	5	30
Processor memory insufficient	1	7	7
Database management software loses derived data	2	2	4

Table 6-2 Risk Exposure Factors for Satellite Experiment Software

CAUTION! Boehm explains there is much uncertainty in subjectively estimating the probability or loss associated with an unsatisfactory outcome, which is in itself, a major source of risk needing to be addressed as early as possible. He says one of the best ways to reduce this source of risk is to buy information that can give you insight into the situation. For instance, to determine whether fault-tolerant features will cause an unacceptable degradation in real-time performance, you can buy information by investing in a prototype. *[Prototyping is discussed in Chapter 14, Managing Software Development.]*

Best Practices Initiative's Risk Management Method

The Program Manager's Guide to Software Acquisition Best Practices, developed under the **Software Acquisition Best Practices Initiative** by the Software Program Manager's Network (SPMN) *[discussed in detail in Chapter 2, DoD Software Acquisition Environment]*, lists ***formal risk management as "Number One"*** of the nine recommended software acquisition best practices. To help program managers with risk management, the SPMN developed the following three-part method:

1. **Address the problem**. All software has risk. The cost of resolving a risk is usually relatively low early on, but increases dramatically as the program progresses.
2. **Practice essentials.**
   - Identify risk.
   - *Decriminalize* risk (i.e., protect the guilty),
   - Plan for risk.
   - Formally designate a **Risk Officer** (a senior member of the management team responsible for risk management). *[Note: This does NOT mean that the RO is the only one responsible for risk! It is everyone's job to watch out for risk and work to manage it.]*
   - Include in the budget and schedule a calculated Risk Reserve Buffer of time, money, and other key resources to deal with risks that materialize.
   - Compile a database for all non-negligible risks.
   - Include technical, supportability, programmatic, cost, and schedule risk.
   - Prepare a profile for each risk (consisting of probability and consequence of risk actualization).
   - Include risks over the full life cycle (not just your watch).
   - Do not expect to avoid risk actualization.
   - Keep risk resolution and workarounds off the critical path by identifying and resolving risk items as early as possible.
   - Provide frequent **Risk Status Reports** to the program manager that include:
     - Top ten risk items,
     - Number of risk items resolved to date,
     - Number of new risk items since the last report,
     - Number of risk items unresolved,
     - Unresolved risk items on the critical path, and
   - Probable cost for unresolved risk versus risk reserve.
3. **Check Status**.
   - Has a Risk Officer been appointed?
   - Has a risk database been set up?
   - Do risk assessments have a clear impact on program plans and decisions?
   - Is the frequency and timeliness of risk assessment updates consistent with decision updates during the program?
   - Are objective criteria used to identify, evaluate, and manage risks?
   - Do information flow patterns and reward criteria within the organization support the identification of risk by all program personnel?

- Are risks identified throughout the entire life cycle, not just during the current program manager's assignment?
- Is there a management reserve for risk resolution?
- Is there a risk profile drawn up for each risk, and is the risk's probability of occurrence, consequences, severity, and delay regularly updated?
- Does the risk management plan have explicit provisions to alert decision makers upon a risk becoming imminent?

As illustrated on Table 6-3, the Network's ***Little Yellow Book of Software Management Questions*** gives a list of questions to help you understand key risk issues and the extent to which best risk management practices are being employed on your program.

	RISK MANAGEMENT PROGRAM QUESTIONS
1	What are the top ten risks as determined by government, technical, and program management staff? When did you last check your top ten risks?
2	How do you identify a risk?
3	How do you resolve a risk?
4	How much money and time do you have set aside for risk resolution?
5	What risks would you classify as show-stoppers and how did you derive them?
6	How many risks are in the risk database? How recently did you update the database?
7	How many risks have been added in the last six months? Describe the most recent risk added to the database? When was it added? What was your mitigation plan for it?
8	Can you name a risk that you had six months ago and describe what you did to mitigate it?
9	What risks do you expect to mitigate or resolve in the next six months?
10	Are risks assessed and prioritized in terms of their likelihood of occurrence and their potential impact on the program? Give an example.
11	Are as many viewpoints as possible (in addition to the program team's) involved in the risk assessment process, Give an example.
12	If you will not remain for the program's completion, what risks have been identified that will remain after you leave? Are any of them imminent? Will you leave a transition plan?
13	Pick a risk and explain the risk mitigation plan for it.
14	What is your top supportability risk?
15	What percentage of risks impact the final delivery of the system? How did you arrive at that decision?
16	To date, how many risks have you closed out?
17	Who in this meeting/briefing is the Risk Officer? Is the role of Risk Officer their primary responsibility? If not, what percentage of the officer's time is devoted to being the program's risk officer?
18	What percentage of your risks have been identified by non-managerial workers? How many risks were identified by the contractor,
19	How are identified risk items given program visibility?

Table 6-3 Risk Management Questionnaire

Team Risk Management Method

The SEI has developed a cooperative risk management method called **Team Risk Management** (TRM) which views the joint government/ industry team approach as a constant process throughout the life cycle. Effective communication about software risk (and systems risk) is addressed from a software perspective. The methods and processes for TRM create effective and continuous risk communications and a positive risk abatement environment within and among government and industry organizations. It also creates a non-threatening atmosphere where risk is raised to higher levels of control, and actions are taken for its mitigation. With this methodology, effective plans can be put in place to manage risk before they become showstoppers.

NOTE: Although it is desirable to have provisions for TRM as part of the contract, the SEI has concentrated their work on building the team environment after contract award and the team is in place.

Team Risk Management Principles

TRM team-oriented activities involve the Government and contractor applying risk management methods together. The seven principles of TRM are illustrated on Table 6-4 (below).

Team Risk Management Advantages

TRM offers a number program advantages as compared to individual or group risk management. It also involves a change from past management practices and government-contractor relationships requiring new commitments by both. These new commitments may involve investment, particularly early in the program. However, one problem prevented can more than pay for the entire investment. Table 6-5 (below) highlights common advantages found on programs implementing TRM. [HIGUERA95]

PRINCIPLE	EFFECTIVE RISK MANAGEMENT REQUIRES
Shared product vision	• Sharing a product vision based upon common purpose, shared ownership, and collective commitment. • Focusing on results.
Teamwork	• Working cooperatively to achieve a common goal. • Pooling talent, skills, and knowledge.
Global perspective	• Viewing software development within the context of the larger systems-level definition, design, and development. • Recognizing both the potential value of opportunity and the potential impact of adverse effects.
Forward-looking view	• Thinking toward tomorrow, identifying uncertainties, anticipating potential outcomes. • Managing program resources and activities while anticipating uncertainties.
Open communication	• Encouraging free-flowing information at and between all program levels. • Enabling formal, informal, and impromptu communication. • Using consensus-based processes that value the individual voice (bringing unique knowledge and insight to identifying and managing risk).
Integrated management	• Making risk management an integral and vital part of program management. • Adapting risk management methods and tools to a program's infrastructure and culture.
Continuous process	• Sustaining constant vigilance. • Identifying and managing risks routinely throughout all phases of the program's life cycle.

Table 6-4 Team Risk Management Principles

ADVANTAGE	DESCRIPTION
Improved communications	By openly sharing risks, both the customer and supplier are able to draw on each other's resources in mitigating risks and enabling rapid response to developing risks or problems.
Multiple perspectives	Bringing both customer and supplier together in mitigating risks opens doors to strategies that both can do together, but neither could do alone.
Broader base of expertise	The combination of customer and supplier brings together a richer pool of experience in perceiving and dealing with risks.
Broad-based buy-in	Risks and mitigation strategies are cooperatively determined by the team (customer and supplier), so all accept the results of the process. Second guessing and criticism after the fact are eliminated.
Risk consolidation	Structured methods bring together risks identified in each organization, giving decision makers a more global perspective and highlighting areas of common interest and concern.

Table 6-5 Advantages of Team Risk Management

Team Risk Management on the NALCOMIS Program Example

During the movie Top Gun, you may have noticed sailors in green flight deck vests on the carrier deck scurrying to prepare Tom Cruise's **F-14 Tom Cat** for launch. These were the unheralded, but much appreciated, squadron maintenance technicians who service and repair Navy and Marine aircraft in organizational maintenance activities (OMAs). According to **Captain Wayne Rogers**, Program Manager of the **Naval Aviation Logistics Command Management Information System (NALCOMIS),** the objectives of his program are to make maintenance personnel's job easier, and to be a combat force multiplier, as well as business transaction cost reducer. NALCOMIS for OMAs is the information *deck plate* that maintainers use in support of Navy and Marine Corps aviators.

As illustrated on Figure 6-11, the NALCOMIS OMA is a management information system (MIS) developed to support aircraft maintenance and material management functions. It received **MAISRC** approval for Milestone III full deployment in December 1994. NALCOMIS OMA was developed with production baseline system IOC achieved in less than three years. Today, over one third of Navy and Marine Corps squadrons use the NALCOMIS OMA.

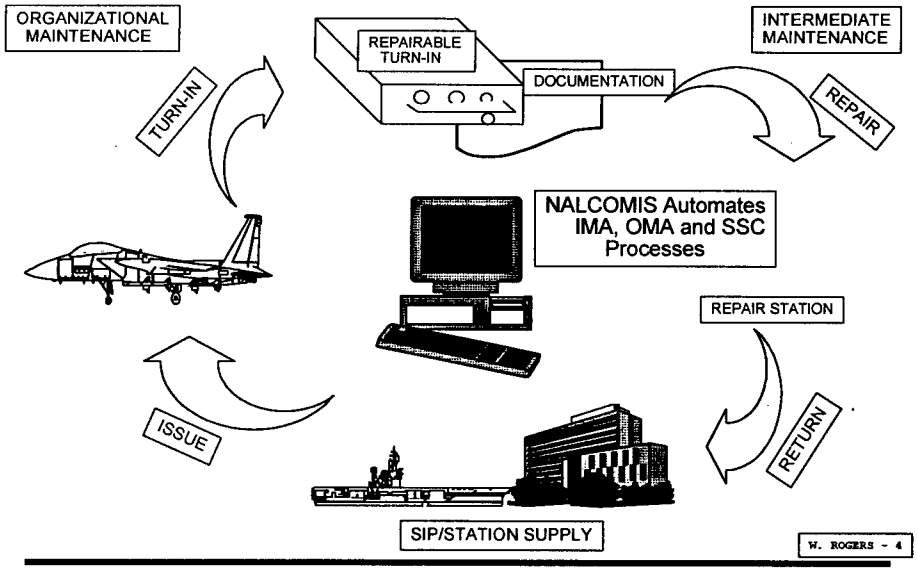


Figure 6-11 NALCOMIS Local Repair Cycle [ROGERS95]

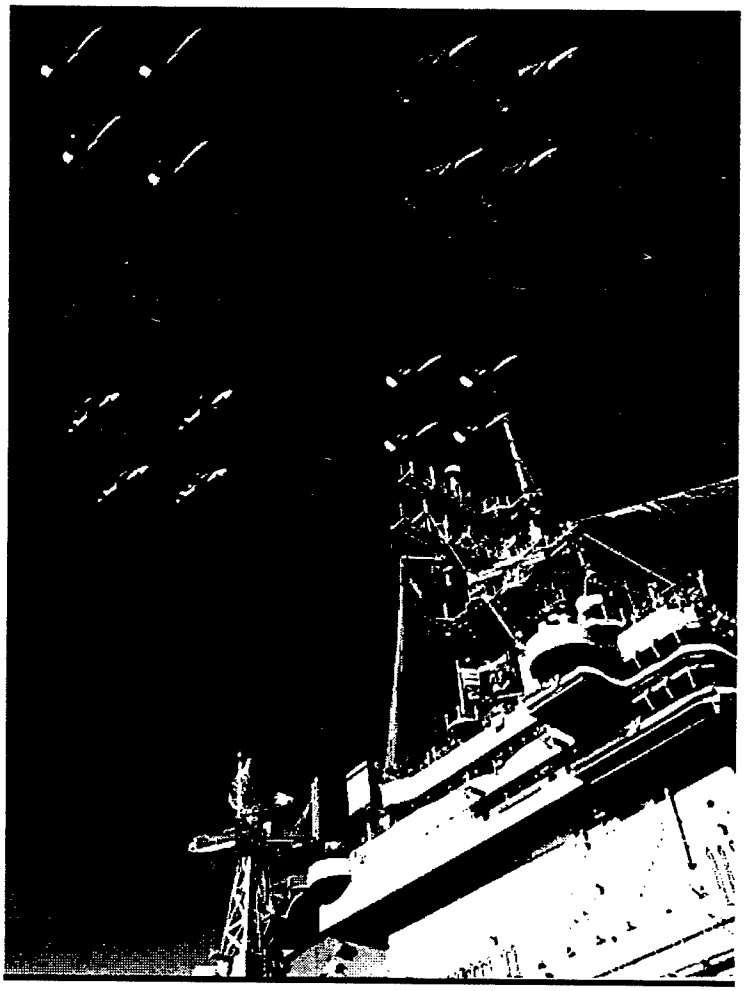


Figure 6-12 One Third of All Navy/Marine Squadrons Use NACOMIS OMA

The program's management approach featured the use of integrated program teams (IPTs) empowered with self-management to achieve goals and deliver products in support of program cost, schedule, and performance guidelines. Risk management was accomplished through periodic review and assessment of key metrics by the Program Office, the **Navy Management Systems Support Office (NAVMASSO)** and program IPTs. The 20% critical software issues that take 80% of the developer's time were constantly prioritized to achieve a timely production baseline. Weekly NAVMASSO release planning meetings were held to review development status.

Development status meetings in which the user partook provided true participative risk management. The users, full members of the development team, dealt first hand and at all levels with all facets of the development, including internal testing, documentation development, training preparation, in addition to code development. This provided a unique insight into how cohesive development tradeoff decisions were made among the entire team. This aided effective communication, one of the hardest, but most important, development tasks. Software development remains a fragile process that is part systems engineering and part human resource management. People management was 80% of the challenge in the NALCOMIS OMA development. Using an integrated program team approach built team consensus and led to integrated technical, business and oversight approaches.

B-1B Computer Upgrade Risk Management Method and Example

Since it's inception as the B-1A bomber, the B-1 has been the subject of negative press coverage. When **President Ronald Reagan** made the decision to produce the B-1 as the low-level penetrator B-1B variant in October 1981, the barrage of negative coverage intensified. The majority of negative press coverage has focused on avionics problems, which have largely been the result of software problems. In June 1992, the Air Force called for new conventional roles for the primarily nuclear deterrent bombers, the B-1B and the B-2A. According to **First Lieutenant Daniel Stormont,** the B-1B will require a systematic series of upgrades over the next fourteen years, nearly all of which will require the development of additional weapons delivery software. Managing the upgrade of the B-1B's computers and associated software, will require judicious risk management procedures in all phases of the upgrade program.

Identified Risks

Many risk management lessons were learned on the B-1B upgrade program. One risk, true for any acquisition program, is ***funding***. Make sure your *limited available funding is not squandered on mistakes or misdirections*. This requires a **good, well-defined set of requirements** and **active management involvement** in the upgrade. Maximize the use of COTS technology to help reduce program cost risk.

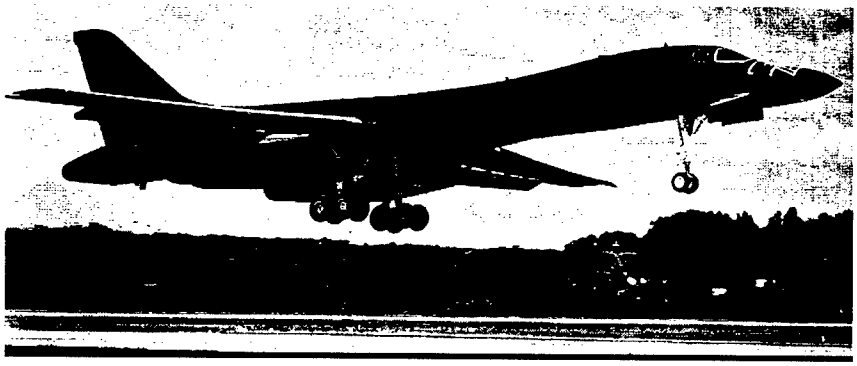


Figure 6-13. Risk Management Filters Through All B-1B Upgrade Phases

Another risk item is the recent initiative from **Secretary of Defense William J. Perry** *[see Chapter 2, DoD Software Acquisition Environment].* The B-1B upgrade is one of the first programs to implement this policy. Essentially, the directive requires the use of best commercial practices wherever possible and to apply MilSpecs only when no suitable commercial standard exists. In some cases, this is very simple to do: the military standard for the Ada programming language (MIL-STD-1815A) is also an ISO standard. In other cases (i.e., DoD-STD-2167A), equivalent commercial standards and practices were (at that time) difficult to identify. *[See Volume 2, Appendix D for a list of government and commercial software standards.]* The issue is made more difficult by the fact that many companies bidding on military contracts do not do commercial work as well (at least not in the same division), thus they frequently are not familiar with current commercial practices.

Finally, the computer upgrade is still required to comply with the (then) appropriation public law mandating the use of the Ada programming language. If modifications to existing code and new code exceed 33% of the size of the CSCI, then the existing JOVIAL code must be converted to Ada. A new processor that cannot support the MMP ISA will also require conversion of the code to Ada. This is a good step to take from a supportability standpoint, but it does require regression testing and recertification, especially for terrain-following and nuclear software.

The best teacher is experience. Therefore, the B-1B Conventional Weapons team has tried to apply lessons-learned during the initial procurement of the B-1B. The lessons most appropriate to the current upgrade include:

- You can never have enough spare memory and throughput;
- Never rely on contractor claims that any firmware item will not require change over the system's life cycle; and
- Conducting a flight test program with a fly/fix/fly strategy is very inefficient.

Contractor Risk Management Teams

One of the tasks the prime contractor assumed was to set up and administer a risk management program. Their approach has been to set up a system of integration control teams (government/contractor) to identify and manage risk areas, as illustrated on Figure 6-14. In addition there are risk management efforts to identify and track risks at the product level by the product teams and at the systems level by the Systems Engineering Team.

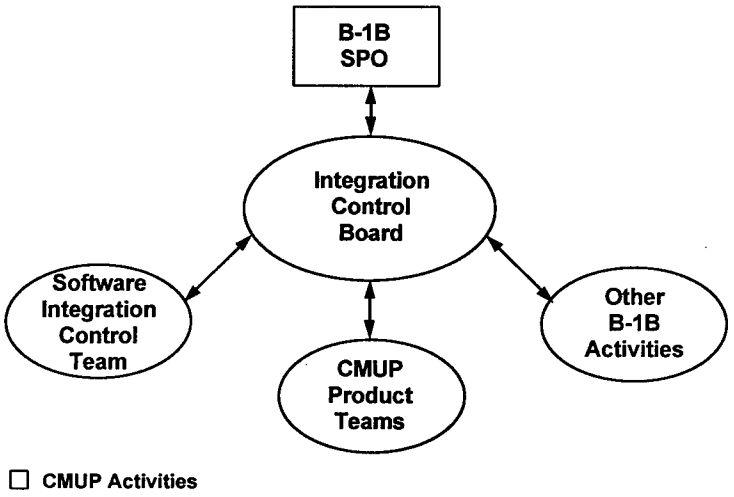


Figure 6-14 Contractor Risk Management Teams [STORMONT95]

Program Office Estimate

An important task in the process of getting CMUP on contract was the **Program Office Estimate (POE).** This required an in-depth cost and schedule analysis of each major task in the program. For the computer upgrade, the software was modeled using the PRICE-S parametric estimation model *[discussed in Chapter 8, Measurement and Metrics]* and was validated against the past performance of the

contractors on sustainment efforts. The hardware was estimated using module-level development and production costs for the **F-15 VHSIC Central Computer (VCC)**, since this computer was considered a good representation of a *typical* system currently in the field. Life cycle costs were also estimated for both hardware and software. In addition to allowing the SPO to prepare the most accurate budget request possible, this estimate helped identify areas of potential cost and schedule risk.

Integrated Risk Management Process

The **Integrated Risk Management Process (IRMP)** is a new Aeronautical Systems Center (ASC) initiative. The B-1B CMUP was the first program to successfully incorporate the IRMP as part of the contracting process. The IRMP extends and augments the program office estimate discussed above by combining it with independent potential risk area evaluations. Experts from the engineering and financial communities at ASC were brought in to take an independent look at the technical content, budget, and schedule of the program. The resulting assessment was provided to the B-1B SPO. Members of the SPO then worked with the evaluators to mitigate risk areas identified in the IRMP. This integrated process provided a vehicle for honest evaluations and analysis of program risk without the *finger-pointing* that frequently occurs when an outside agency evaluates a SPO's efforts. The assessment from the IRMP identified several areas requiring the SPO's attention which were addressed before CMUP contract award.

Chief Engineers' Watchlist

The B-1B program is using a new spin on the old *watchlist* technique. While risk item lists are being maintained by the product and systems engineering teams, this list allows contractor chief engineers and the SPO to identify and track risk items of particular interest or importance. These items may not be on any other watchlists. Often the product teams or the systems engineers at the working level may not recognize a risk apparent to technical personnel at the management level. This list ensures these items are not overlooked.

Computer Resources Working Group

One of the most important risk management tools available to the B-1B SPO is the **Computer Resources Working Group (CRWG)** meetings held quarterly in Oklahoma City. *[The B-1B is supported by the Oklahoma City Air Logistics Center at Tinker AFB.]* In addition to the task of maintaining the **Computer Resources Life Cycle Management Plan (CRLCMP),** the CRWG provides a forum for in-depth technical discussions among software developers, hardware vendors, Air Combat Command (ACC), SPO members, and interested outside agencies [i.e., Air Force Operational Test and Evaluation Center (AFOTEC), discussed in Chapter 14, *Managing Software Development*]. Open communications between all B-1 community members are encouraged in these meetings. Issues can be identified and frequently resolved at the working level through the CRWG. Any issues that can not be resolved at the working level are brought to the attention of upper management at the executive outbrief. The CRWG has been the B-1B SPO's most valuable tool for software-intensive systems risk management. [STORMONT95]

RISK MANAGEMENT PLANNING

It is important to be aware of your software risk factors. It is also important to know about risk management methods and have one implemented on your program. However, the real key to software risk management is planning and implementing your plan. **Risk management planning** is an on-going effort to identify all significant risk probabilities and take action to minimize their occurrence and/or impact. Planning prevents most risks from becoming problems. If you do not know your risks and their potential impacts, you are planning to let the risks fester. Furthermore, the events used as decision points in a contracted effort will have greater meaning if they correspond to those points when important program uncertainties become known. Good management not only entails reasonable estimation, it means leaving nothing to chance (or at risk) that you can control through planning. Keeping an historical perspective on what has and has not worked for programs similar to yours is vital. East Roman Emperor and general, Maurice, having defeated the Persians and Avars circa the year 600AD, taught us that

> *In battles and in every action against the enemy, the wise general, even the most courageous, will keep in mind the possibility of failure and defeat and will plan for them as*

actually occurring... The sharp general takes into account not only probable dangers, but also those which may be totally unexpected. [MAURICE600AD]

Risk Management Plan

The **Risk Management Plan (RMP)** is a controlling document that states how risk analysis and procedures are applied to your program. It describes all aspects of the risk identification, estimation, evaluation, and control process. The initial part of this plan is the **Risk Estimate of the Situation (RES),** based on the standard Navy **Commander's Estimate of the Situation (CES).** [NAVY95] The RES clearly identifies four software-intensive program elements:

- The objectives (measurable and controllable program goals);
- The strategies (broad constraints or rules under which the goals of the program can be met);
- The tactics (the specific actions, the *what* that happens during a given situation); and
- The resources (constrain the tactics by dictating to use).

Each of these elements are spelled out in the **Software Development Plan (SDP)** *[discussed in Chapter 14, Managing Software Development].* Grouping each of these element descriptions together aids in determining the variables and environment, both technical and nontechnical, in which the system under development's is to operate. Early identification of false efforts and quick recognition of missing ones — both precursors to schedule delays and cost overruns — are the primary RES goals. The RES helps us understand how each element interacts with and affects other elements so we can determine how they contribute to overall program success criteria. [CHARETTE89]

The second part of the RMP is the **Risk Aversion** (or mitigation) **Plan (RAP),** a detailed plan for risk resolution after the general risk analysis process has been conducted. For each identified risk, it states what actions (i.e., risk avoidance, control, assumption, or transfer) will be taken for its mitigation. The RAP should include, as a minimum:

- A breakdown of all program risk areas and representative risk factors in each area;
- Identification of priority risk items with a ranking of importance in relation to program objectives;
- Identified alternatives and their costs;
- Recommended mitigation strategy for each risk item including action plans for each (risk items requiring no action should also be noted);
- Integration strategy for individual risk aversion plans (with attention to combining action plans for more than one risk item);
- An integration strategy for the RES, RMP, and RAP;
- Assignment of resources needed to implement the risk aversion strategy which includes cost, schedule and technical considerations;
- Description of the Risk Management Team, its responsibilities, how it fits into the total program, and the person to be assigned the Risk Champion;
- Implementation start date, schedule, and key milestones;
- Criteria for success (i.e., when will the risk be considered mitigated) and the monitoring approach to be used;
- Tracking, decision, feedback points, and monthly reports, including when the priority risk item list and plans are to be updated;
- Contingency plans for each prioritized risk; and
- Upper management sign-off of the RAP. [CHARETTE89]

Thus, your RMP consists of an integrated approach to each risk item you identify. It should be based on answering the standard questions of *why, what, when, who, where, how,* and *how much*. For example, if you plan to buy information by building a prototype on fault-tolerant features, that same prototype may be used to reduce latent software defect uncertainties or to involve the client in interface design. Also, your RMP must be thoroughly integrated into your SDP. If you have identified the need for a 10-week prototype development and demonstration period, that time must also be added to the overall program schedule so it is current and realistic. *[See Chapter 12, Strategic Planning, for a discussion on schedule and cost estimating.]*

Contingency Planning

Napoleon's Military Maxim Number 8 states:

> *A general-in-chief should ask himself frequently in the day, "What should I do if the enemy's army appeared now in my front, or on my right, or on my left?" If he has any difficulty in answering these questions he is ill posted and should seek to remedy it.* [NAPOLEON31]

Contingency planning addresses those risks that require monitoring for some future response should the need arise. **Roetzheim** gives an example of when he was a program manager on a contract to develop a software-intensive system for the Navy. The system was to graphically plot specific target positions by monitoring tactical Navy data link circuits. He was told the Government would supply the software interface (to be designed by government personnel) to convert the existing electrical signals into a compatible format with his system. After interviewing the government design team, Roetzheim was convinced they did not have the necessary experience in the specialized Navy circuitry to successfully perform the task. He, thus, concluded that the probability of failure for this critical item was fairly high. The situation was further compounded by the fact that his software program was on a tight schedule, with final delivery culminating in an at-sea demonstration during a major fleet exercise. In short, the consequences of failure for his program were extreme. One month prior to the full-scale demonstration, his worst fears were confirmed. The government-supplied board failed dismally during the first live test run. Luckily, his contingency planning included the following actions:

- He had company engineers rough out an initial interface design in case company in-house development was required;
- Based on the initial design, the company archives were searched and all required technical data and specialized parts were located and on-hand;
- Company individuals capable of performing the task were identified and their availability was confirmed if required for an emergency job;
- Overtime for the engineering department was approved by management; and
- A development plan, including all cost estimates and a statement of work, were prepared and all the information needed to write a delivery order was ready.

The afternoon of the live test run failure, Roetzheim convinced his Navy contracting officer to allow him to begin work on the interface in parallel with the government staff. Although, as feared, the government-furnished board never did work, the at-sea demonstration was successfully completed using the in-house developed interface. If he had not identified this high-risk item and performed adequate contingency planning, his software program would have been a high-profile, embarrassing fiasco — damaging his reputation and the possibly of future work for his company. [ROETZHEIM88]

Fairley uses an example of late hardware delivery risk. His contingency plan was to monitor the hardware vendor's progress, while developing a software emulator of the risky component. When contingency planning, you must also justify the added cost of preparing a contingency plan, monitoring the situation, and implementing action plans. If the cost is justified (as in Roetzheim's case where the cost of failure was extreme), plan preparation and vendor monitoring might be implemented immediately. Whereas, the action to develop an emulator might be postponed until the risk of late delivery becomes a real threat (i.e., the vendor's schedule slipped beyond some predetermined unacceptable threshold.) This, Fairley explains, brings up the issue of adequate lead time. When should you start to develop the emulator? The answer comes from early analysis of the probability of late delivery and its consequences. As the probability becomes greater, the urgency to build the emulator becomes greater. With proper contingency planning, you will establish a drop-dead date, set aside funds and personnel to implement your plan, while still staying within your preplanned delivery schedule. [FAIRLEY94] Contingency planning involves the following:

- Specifying the nature of the potential risk;
- Considering alternative approaches;
- Specifying constraints;
- Analyzing alternatives; and
- Selecting an approach.

The **Contingency Plan** includes:

- Risk factors;
- Tracking methods;
- Responsible parties;
- Thresholds;
- Resource allocations; and
- Constraints. [FAIRLEY94]

Crisis Management Plan

Contingency planning involves preparing a Contingency Plan which includes a **Crisis Management Plan** and a **Crisis Recovery Procedure**. Contingency Plans address risks not identified in the RMP action plans discussed above. The Crisis Management Plan is the backup plan used if the Contingency Plan fails to resolve the risk within a specified time. The Crisis Recovery Procedure is invoked when the crisis is over, whether it had a positive or negative outcome.

A crisis is an overall ***show-stopper***! All program effort and resources must be focused on resolving the crisis situation. Once in crisis, you must muster your forces, go on the offensive, and ***attack!*** Because if you do not attack this type risk, it will attack you and win! As Frederick the Great told his generals before battle,

> *Gentlemen, the enemy stands behind his entrenchments, armed to the teeth. We must attack him and win, or else perish. Nobody must think of getting through any other way. If you don't like this, you may resign and go home.* [FREDERICK47]

A crisis occurs when your Contingency Plan fails to resolve an unforeseen event. If you do not act quickly to manage a major unforeseen negative event, you may as well resign and go home. This would have happened if perhaps Roetzheim were unable to allocate enough engineers or hours to develop the interface. Once the government-supplied one failed, he would have been out of time, over budget, and still not have the board he needed to complete the at-sea demonstration successfully. Before a crisis materializes, you may be able to define some elements of crisis management, such as the responsible parties and a drop-dead date, but you may be hard pressed to plan the exact details until the crisis occurs. Fairley explains what you must do in such a situation:

- Announce and publicize the problem;
- Assign responsibilities and authorities;
- Update status frequently;
- Relax resource constraints (fly in experts, bring on emergency personnel, provide meals and sleeping facilities to keep people on site until the crisis is resolved, etc.);
- Have program personnel operate in burnout mode;
- Establish a drop-dead date;

- Clear out unessential personnel. [FAIRLEY94] *[Also see Chapter 16, The Challenge, for a discussion on "What To Do With a Troubled Program."]*

Crisis Recovery Plan

Once recovered, you must examine what went wrong, evaluate how your budget and schedule have been affected, and reward key crisis management personnel. During crisis recovery, you should:

- Conduct a crisis postmortem, fix any systematic problems that caused the crisis, and document lessons-learned; and
- Recalculate cost and time to complete the program, rebaseline, and update your schedule and cost estimates to reflect these new projections. *[See Chapter 12, Strategic Planning.]*

RISK ELEMENT TRACKING

Another important function is risk element tracking. **Risk element tracking** involves identification of your program's highest-risk issues and tracking progress towards resolving those issues through subsequent progress reports. The major risk management benefits are similar to those of cost/schedule/performance tracking plus the added ones of identifying and maintaining a high-level risk consciousness. Tracking becomes critical because the one risk attribute whose influence is difficult to predict is ***"time."*** Generalizations about risk made early in the program can (and often do) decay with time. One reason for performing risk tracking is to keep a predictable, unpredictable, or unknown risk from becoming a known one. Tracking occurs after the decisions about mitigation strategies and tactics have been implemented to:

- Check if the consequences of our decisions are the same as envisioned;
- Identify opportunities for refinement of the RAP; and
- Help provide feedback for future decisions about controlling those new or current risks not responding to risk mitigation or whose nature has changed with time. [CHARETTE89]

Risk Tracking Methods

As you will learn in Chapter 8, *Measurement and Metrics*, the most effective way to track risk is **measurement**. The significant risk reducing benefits achieved through a comprehensive measurement program cannot be overstressed. Metrics data provide the means to compare your risk elements with historical data, pinpoint risk drivers, and determine alternative risk reduction choices. You must aggressively track and control the risk drivers affecting your program.

Cost/schedule/performance tracking involves using techniques such as WBSs, metrics, quality indicators, activity networks, **earned-value *methods*** *[discussed in Chapter 15, Managing Process Improvement]* to determine and track program progress with respect to plans, schedules, and budgets. Cost/schedule/performance tracking is useful because potential schedule slippages, cost overruns, and performance shortfalls are identified early, and their impact on other interdependent system elements reduced. Other risk tracking methods are peer inspections, reviews, audits *[also discussed in Chapter 15]*, and CRWG meetings *[discussed above]*.

EXAMPLE: **One highly successful company, never more than 10% above or below predicted cost and schedule, established a Risk Management Database. For each risk it assesses impact (high, medium, low) and probability of occurrence (high, medium, low), and forces managers and engineers to agree on dollar and schedule impacts for each risk. For each program, each month, impact and probability (including quantified cost and schedule impacts) risk levels are revisited. The database is kept current, with the risk analysis data and mitigation plans. The status of risks and their actions, corrections of RAP deviations, and dissemination of risk management knowledge to all affected parties, including senior management, is tracked and monitored.**

ADDRESSING RISK IN THE RFP

Andy Mills, from the US Army Communications and Electronics Command (CECOM) Software Engineering Directorate, has defined a risk-based source selection approach where offerors' proposed technical approaches are evaluated and awarded based on how effectively they address software risk management. In your RFP, he suggests you incorporate sufficient emphasis on both proposal risk and offeror's performance risk to ensure risk will be managed throughout the acquisition. The offeror's proposed approach must provide a step-by-step detailed execution plan, as well as a detailed organization, process, technology, and design strategy. To aid in the evaluation process you also need to employ your own risk management methodology from a Government perspective. A standard means to evaluate risk simplifies the evaluator's job, provides a more thorough evaluation, and increases the effectiveness of acquisition streamlining. *[See Chapter 2, DoD Software Acquisition Environment, for a discussion on acquisition streamlining.]*

Offeror's Risk Methodology

The effectiveness of acquisition streamlining depends on the establishment of a planned acquisition path which manages risks. By requiring a risk-based approach, offerors' proposals should state how they will plan and schedule software activities based upon realistic assessments of technical challenges and risks. They should describe how they plan to attack software risks through an appropriate choice of software architectures, reuse strategies, requirements management processes, metrics, development models, tools, and technologies. All these are elements must be addressed in offerors' proposals. In fact, the offeror's proposal is the initial plan for the software development. Subsequent adjustments of the approach often become necessary, as the focus of program risk shifts during development. However, a generic software approach that does not plan for program and product risks, will surely result in having to resolve otherwise foreseeable problems when resources are already spent or committed elsewhere.

Risk-Based Source Selection

Proposal evaluation has traditionally depended on the opinions of teams of highly-qualified technical personnel participating in the source evaluation and selection process. Using a software risk evaluation methodology extends the capabilities of evaluation personnel by facilitating risk identification and by focusing attention on the feasibility and merits of each offeror's approach.

To understand what is meant by requiring a proposal based on risk management, assume that every software development activity is traceable to the mitigation of one program risk or another. An offeror's proposal should organize main software development risks by building their approach on exploiting opportunities for their mitigation whenever possible. Table 6-6 provides a typical list of proposal areas. These areas represent possible *solution areas* for mitigating software development risks. From industry's perspective, the proposal tells the Government why the proposed approach is the best possible. For each element of the proposal, industry should point out in what manner the proposed activity contributes to risk reduction.

PROPOSAL ITEMS	
Difficult requirements	Risk management implementation
Software architecture	Software tracking
Domain reuse	Technical reviews
System capacity and resources	Subcontractor management
Software engineering environment	Program organization and resources
Software technologies	Personnel
Proprietary nature/Government data rights	Software work breakdown (WBS)
Government furnished items	Engineering procedures
Process improvement	Planning and replanning - Identifying the software activities - Estimating activities - Scheduling software activities - Activity network
Testing	
Corrective action process	
Configuration management	
Software quality evaluation	
Preparing for software transition	

Table 6-6 Proposal Items or Solution Areas [MILLS95]

The first step is really not that far from current practice. Generally feasibility and risk are firmly addressed in proposals at present. However, the purpose of organizing the proposal around risk is to facilitate obtaining a well-planned software development approach which can be readily evaluated for its strength in the mitigation and management of program risks. A systematic method for risk evaluation

is a logical second step. Current source selection teams perform an evaluation on each part of the technical proposal in a systematic manner, one step at a time. Additional risk identification should be employed, such as the SEI's Software Development Risk Taxonomy, as illustrated on Table 6-7, and associated Taxonomy Based Questionnaire (TBQ). The TBQ addresses source selection and poses critical questions within each of the detailed taxonomy attributes which are intertwined within risk reduction and management activities for each taxonomy topic. This permits assessing the feasibility of a proposed approach in detail without prescribing required activities or practices.

PRODUCT ENGINEERING	DEVELOPMENT ENVIRONMENT	PROGRAM CONSTRAINTS
Requirements	**Development Process**	**Resources**
Stability	Formality	Schedule
Completeness	Suitability	Staff
Clarity	Process control	Budget
Validity	Familiarity	Facilities
Feasibility	Product control	**Contract**
Precedent	**Development System**	Type of contract
Scale	Capability	Restrictions
Design	Suitability	Dependencies
Functionality	Usability	**Program Interfaces**
Difficulty	Familiarity	Customer
Interfaces	Reliability	Associate contractors
Performance	System support	Subcontractors
Testability	Deliverability	Prime contractor
Hardware constraints	**Management Process**	Corporate management
Non-developmental software	Planning	Vendors
Code and Unit Test	Program organization	Politics
Feasibility	Management experience	
Testing	Program interfaces	
Coding	**Management Methods**	
Implementation	Monitoring	
Integration and Test	Personnel management	
Environment	Quality assurance	
Product	Configuration management	
System	**Work Environment**	
Engineering Specialties	Quality attitude	
Maintainability	Cooperation	
Reliability	Communication	
Safety	Morale	
Security		
Human Factors		
Specifications		

Table 6-7 Taxonomy of Software Development Risks
[CMU/SEI-93-TR-6]

To realize the benefits of risk-based acquisition we must permit industry to freely determine and apply best practices to leverage industrial competitiveness. Beyond past performance, strong proposal risk evaluation helps ensure that software developers impose sufficient process requirements and discipline upon themselves. For industry, a risk-based strategy provides a long awaited return on investment in corporate and enterprise capabilities. A risk-based methodology enables the use of non-government practices where and when they make sense for defense programs and gives an increased incentive to industry to submit proposals which can be managed without extensive oversight. A risk-based RFP enables better management of program risk, a more successful program, increased use of modern technical approaches, less documentation expense, effective partnering, and the benefits derived from increased program visibility and insight. [MILLS95]

Performance Risk Analysis Group (PRAG)

The PRAG must be staffed by senior-level individuals with the competence and experience necessary to clearly assess offerors' competence and the relevancy of the offerors' past contract performance as it compares to the efforts required in your RFP. Too often we have assigned these duties to junior personnel who lack the experience to make these critical judgments. For past experience to impact the source selection process, the data collected must be timely, relevant to the proposed acquisition, and useful to the decision-maker. Therefore, the first step in a successful PRAG is to select the appropriate senior people to staff it.

PRAG members must never accept at face value the *write-ups* they obtain from Contractor Performance Assessment Reports (CPARS) or questionnaires. They must, in all cases, ensure that they have personally contacted the government program managers and contracting officers to obtain the right level of detail to fully understand the type of effort performed under a reported contract and the rating. The PRAG needs first hand, accurate information to help ascertain relevancy and specific performances. PRAG members must also investigate and report to the Source Selection Authority (SSA), *cause and effect* of rating changes and any other inconsistencies in the data. The PRAG team and the quality of their analysis can be the difference between a good SSA decision and a mistake.

SOFTWARE RISK MANAGEMENT BEGINS WITH YOU!

This chapter has identified the need for risk management in the software acquisition and engineering discipline. It has also identified software risk factors, techniques and processes for managing risks, and formal methods you should consider for application to your program. This chapter further described the importance of risk management planning, risk element tracking, and addressing risk in the RFP. In short, you have the necessary information to implement software risk management on your program. It's now up to you to identify your software risk factors, and take appropriate steps to implement formal methods to manage software risk. ***Software risk management begins with you!***

REFERENCES

[AUGUSTINE83] Augustine, Norman R., *Augustine's Laws*, American Institute of Aeronautics and Astronautics, New York, 1983

[BLUM92] Blum, Bruce I., Software Engineering: A Holistic View, Oxford University Press, New York, 1992

[BOEHM91] Boehm, Barry W., "Software Risk Management: Principles and Practices," *IEEE Software*, January 1991

[CHARETTE89] Charette, Robert N., Software Engineering Risk Analysis and Management, McGraw-Hill Book Company, New York, 1989

[CHURCHILL99] Churchill, Sir Winston, The River War: An Historical Account of the Reconquest of the Sudan, 2 Volumes, Longmans, Green, and Company, London, 1899

[EVANS94] Evans, "Thread of Failure: Project Trends That Impact Success and Productivity," *NewFocus,* Number 203, Software Program Managers Network, Naval Information System Management Center, March 1994

[FAIRLEY94] Fairley, Richard, "Risk Management for Software Projects," *IEEE Software*, May 1994

[FREDERICK47] Frederick II the Great, "Instructions to His Generals," 1747, included in T.R. Phillips, editor, Roots of Strategy, Book 1, Stackpole Books, Harrisburg, Virginia 1985

[HALL95] Hall, Elaine M., and F. Carol Ulrich, "Streamlining the Risk Assessment Process," paper presented to the 7th Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[HIGUERA95] Higuera, Ronald P., "Team Risk Management," *CrossTalk*, January 1995

[LATIMES88] "B of A's Plans for Computer Don't Add Up," *The Los Angeles Times*, February 7, 1988

[LEE33] Lee, GEN Robert E., as quoted by J.F.C. Fuller, Grant and Lee: A Study in Personality and Generalship, Eyre and Spottiswoode, London, England, 1933

[MARCINIAK94] Marciniak, John J., Editor in Chief, Encyclopedia of Software Engineering, Volume 2, John Wiley & Sons, Inc., New York, 1994

[MARSHALL41] Marshall, GEN George C., address to the first officer candidate class, Fort Benning, Georgia, September 18, 1941

[MAURICE600AD] Maurice, Flavius Tiberius, The Strategikon, circa 600AD.

[MILLS95] Mills, Andy, "Software Acquisition Improvement: Streamlining Plus Risk Management," paper presented to the Seventh Software Technology Conference, Salt Lake City, Utah, April 1995

[NAPOLEON31] Napoleon Bonaparte I, The Military Maxims of Napoleon, 1831, David Chandler, editor, George C. D'Aguilar, translator, Greenhill Books, London, 1987

[NAPOLEON55] Napoleon Bonaparte, as quoted by Christopher J. Herold, editor, The Mind of Napoleon: A Selection from His Written and Spoken Words, Columbia University Press, New York, New York, 1955

[NAVY95] Taught in the Employment of Naval Forces course, "The Military Planning Process," US Naval War College, Newport, Rhode Island, 1995

[PRESSMAN93] Pressman, Roger S., "Understanding Software Engineering Practices: Required at SEI Level 2 Process Maturity," Software Engineering Training Series briefing presented to the Software Engineering Process Group, July 30, 1993

[ROETZHEIM88] Roetzheim, William H., Structured Computer Project Management, Prentice Hall, Englewood Cliffs, New Jersey, 1988

[ROGERS95] Rogers, CAPT Wayne, (USN), "A DoD Incremental Development Success Story: NALCOMIS OMA," paper presented to the Seventh Software Technology Conference, Salt Lake City, Utah, April 1995

[STORMONT95] Stormont, 1st Lt Daniel (USAF), "Risk Management for the B-1B Computer Upgrade," paper presented to the Seventh Software Technology Conference, Salt Lake City, Utah, April 1995

CHAPTER 7

Software Development Maturity

CHAPTER OVERVIEW

> *The development of a weapon system requires integrating technical, administrative, and management disciplines into a cohesive, well-planned, and rigorously controlled process. As a critical component of a weapon system, software must be developed under a similarly disciplined engineering process.* [DSMC90]

> *Software is like entropy. It is difficult to grasp, weighs nothing and obeys the Second Law of Thermodynamics; i.e., it always increases.* [AUGUSTINE86]

The above statements sum up the need for ***maturity*** *in software development — we must have a disciplined, engineering approach to software development, while maintaining a grasp on the process and product. If we cannot achieve either of these, we have introduced unnecessary risk in meeting customer needs within cost and schedule constraints.*

The Air Force and the Software Engineering Institute (SEI) have each produced a ***software development capability assessment method*** *for use in source selection to determine the maturity and associated degree of risk of a software developer's process. These assessments provide important insight whether the developer has a process that is predictable, repeatable, and manageable in terms of cost and schedule. These should be used for source selection of any organization requiring the development or modification of software — contractor or government. This includes post-deployment software support, since the support organization is usually different (with different development processes) from the original developer.*

To assist developers in assessing their current maturity, as well as to guide process improvement efforts, the SEI developed the ***Capability Maturity Model℠*** *(CMM℠) from which a family of maturity models has evolved. These models are based on* ***key process areas (KPAs)*** *defined a set of minimum requirements needed to achieve a given level of maturity. Knowing KPA requirements, an organization can address those areas*

needing improvement to achieve the next higher level of maturity. This family of maturity models will help you to specify what level of maturity is required of offerors. You should require that they have a mature, well-defined, standardized process for software development and maintenance.

CHAPTER 7

Software Development Maturity

PROCESS MATURITY: An Essential for Success

Webster defines **process** as, *"A series of operations performed in the making or treatment of a product, e.g., a manufacturing process."* A **software process** is the series of operations performed in making or maintaining of a software product. A **software process definition** is the description of that process. The process definition guides teams of software engineers in the performance of their work. Thus, a **defined, disciplined process** is one that is documented, taught, applied, measured, used by everyone in the organization, and accessible to all team members (e.g., an organization's procedures manual). A defined process accomplishes the following:

- It provides the basis for examining and **improving** the software process;
- It aids in establishing **predictability**;
- It improves **understanding** of roles and dependencies;
- It guides software personnel through orderly **decisions**,
- It provides a smooth working **framework**; and
- It helps staff members to readily **transition** from one program to another. [CLOUGH92]

Without a defined, disciplined software process each team member's work rests on intuition and the quality of their product on blind faith. Team members are left to arrive at their own operational processes, methods, procedures, and standards without the direction and support professionals in other disciplines consider essential (e.g., in sports, the arts, or science). As General George S. Patton, Jr., explained,

All human beings have an innate resistance to obedience. Discipline removes this resistance, and by constant repetition, makes obedience habitual and subconscious. Where would an undisciplined football team get? The players react subconsciously to the signals. They must, because the split second required for thought would give the enemy the jump. [PATTON47]

Watts Humphrey explains that a software team without a defined, disciplined process is one where everyone follows their own individual procedures, which he also equates to a ball team where different members are playing their own brand of ball. If some members are playing soccer, some baseball, and others football, the team is ineffective and cannot produce a quality product. In contrast, a team with a precise process definition is one where each individual's work is coordinated and their progress tracked. More explicitly, Humphrey says the **software process** is the technical and management framework for applying engineering methods, tools, procedures, and people to software development, while the **process definition** identifies roles and specifies tasks. The definition also establishes measures and provides entry and exit criteria for every major step in the process. [HUMPREY95]

Software development organizations are more successful when they have processes they can effectively communicate, manage, and evolve. A well-defined process is also easier to improve. For instance, if some steps in the process are skipped, or if the process is inefficient, problems may occur. Steps, or the process itself, may not be used if the definition is poor, communication is unclear, or team members are not motivated. Improvements can be made once these problems are identified. The process, its definition, and the supporting infrastructure all evolve and mature with use and experience. [HUMPHREY95]

SOFTWARE DEVELOPMENT CAPABILITY ASSESSMENT METHODS

Software development capability assessments are an effective method for determining the maturity of an organization's process. They provide a performance rating system that was established to be fair, accurate, and enforce uniform procedures, clear definitions, consistent measurements, and reliable information to keep vendors

from challenging negative ratings. These assessments involve visits to bidders' facilities to determine their readiness to perform on a contract and their software development maturity. They are used to ascertain whether the developer has a mature software process in place that is predictable, repeatable, and manageable in terms of cost and schedule. The purpose of these assessments is risk mitigation *[discussed in detail in Chapter 6, Risk Management]*. They are used to determine what risks are associated with contracting a given organization to perform your development task. An award to a contractor with a mature, well-defined, standardized process can translate into substantially lower program risk and cost savings for the Government through reduced documentation, oversight, review, and auditing requirements.

REMEMBER: In addition to having a mature software development process, the developer should also have experience in the application domain being developed. Although a developer might have a high-level process maturity, the lack of domain expertise could have a drastic impact on product development (i.e., performance, cost and schedule). *You need both process maturity and domain expertise to minimize software development risk.* This chapter addresses development maturity, and assumes the software developer has the necessary domain experience.

Two software development capability assessment methods are available for source selection evaluations. The Aeronautical Systems Center **Software Development Capability Evaluation (SDCE)** is best used for developers of weapon systems with embedded software or any application requiring substantial systems engineering. The SEI **Software Capability Evaluation (SCE)** is appropriate for MIS developers, and has been used for C4I developers; however, with the substantial systems engineering required by C4I programs, you should consider performing a SDCE to ensure the developer has a mature systems engineering capability. The **Software Process Assessment (SPA)** program was also developed by the SEI. *[NOTE: The SPA program has been updated, and is now the **CMM[SM]-Based Appraisal — Internal Process Improvement (CBA-IPI)**.]* The CBA-IPI focuses on helping development organizations (government and contractor) assess and improve their own processes. [HUMPHREY91] The objective of these methods is to provide

structured, consistent, and comprehensive approaches for evaluating the software process to determine the capability of any organization having software development responsibility after contract award. A high rating on the review does not guarantee software development success; but the evaluation does isolate areas needing closer consideration during source selection (and if selected, after contract award).

NOTE: For questions about C3 and ground electronics systems acquisitions capability assessments, contact Electronic Systems Center *[see "Evaluating C3 Systems" in Volume 2, Appendix A].* **For MIS acquisitions contact the Standard Systems Group** *[see "Evaluating MIS Systems" in Volume 2, Appendix A].* **For avionics and embedded systems contact Aeronautical Systems Center** *[see "Evaluating Embedded/ Avionics Systems" in Volume 2, Appendix A].* **Air Force in-house software development organizations with questions on Software Process Improvement and Software Maturity Assessments should contact the Air Force C4 Agency** *[see "Software Process Improvement and Software Maturity Assessments" in Volume 2, Appendix A].*

Once your program is assessed, it is of little or no use if you are not committed to ***unremitting improvement***. No matter how often the assessment is performed, it is only a starting point. Each time an assessment is performed, it identifies your current level of capability — but more importantly — it identifies a point from which to begin your next round of improvement. Those few organizations who have achieved a CMM[SM] Level 3 or above claim they got there, and stay there, because they have an organization-wide quality attitude. Always looking for ways to improve, they develop an extensive set of measures they perpetually re-evaluate.

NOTE: Refer to the February 1994, July 1995, and September 1995 issues of *CrossTalk* (published by the STSC) for timely, pertinent articles on the subject of process improvement and moving up the maturity scale. Copies are available from STSC customer service *(see Volume 2, Appendix A)* **or can be viewed online** *(see Volume 2, Appendix B for Web address).*

Software Development Capability Evaluation (SDCE)

As described in AFMC Pamphlet 63-103, the **SDCE** evaluates a contractor's ability to develop software for a specific weapon system program, as defined in the RFP. It also helps to decide whether the contractor has the capacity and sufficient qualified personnel available to complete the proposed software development. Assessing capability during source selection accomplishes three related objectives:

- The offeror's capability and capacity to develop the required software within the program baseline is determined;
- The review process elicits a contractual commitment by the offeror, if selected, to implement the methods, tools, practices, and procedures making up their software development process; and
- Insight is gained into each offeror's systems and software engineering development methods and tools to be applied to your program.

The SDCE concentrates on five areas: management approach, management tools, development practices, personnel resources, and Ada technology. The review is normally performed during the EMD RFP preparation and source selection acquisition phase. However, when software developed during Dem/Val is planned to be carried through to EMD, an SDCE should be performed during the Dem/Val source selection phase. Your RFP must state that offerors provide specific information describing their software development methods, including examples of how their methods have been applied on past or on-going programs. If an open discussion is conducted, the an in-plant review of the offeror's team is performed by the Government. The evaluation can also be based solely on the material submitted with the proposal, with the in-plant portion of the SDCE conducted after contract award. *[Aeronautical Systems Center policy requires the use of SDCE results in all weapons systems software source selections.]*

SEI Software Capability Evaluation (SCE)

The **SCE** is described in *SCE Method Description* (CMU/SEI-94-TR-06, Version 2.0, June 1994) and the *SCE Implementation Guide,* (CMU/SEI-94-TR-05, Version 2.0, February 1994) by the **SEI.** The *SCE Method Description* contains an appendix of *"look-fors"* and guidance in determining contractor strengths and weaknesses against **key process areas (KPAs)**. *SCE Implementation Guide* contains suggested wording for acquisition documents. *[See Volume 2, Appendix M, for more information.]*

The SCE assesses contractors' capabilities in four areas: (1) organization and resource management, (2) software engineering process and management, (3) tools and techniques, and (4) software development expertise. The SCE specifically analyzes multiple programs for a discrete acquisition and reports findings at the organizational level for a vendor site based on the discrete programs evaluated. [BARBOUR93] The SCE is used in source selection, and contractors without mature, well-defined, standardized processes are considered high risk. Also, contractors can use this evaluation for self-assessment of their current maturity level and as guidance for achieving higher levels.

A WORD OF CAUTION! An SCE investigates areas generally limited to the processes used. For example, this includes the process of selecting appropriate tools and methods, and training personnel to use them. However, an SCE does not evaluate whether the processes themselves are effective or efficient, nor does it address the appropriateness of the tools and methods used by the developer. Therefore, a proposal by a mature software development organization to use new, state-of-the-art tools and methods could be a significant risk if the developer does not have an experience base to handle them. [HOROWITZ95]

SEI transition partners train source selection teams on conducting SCEs. SEI instructor personnel do not lead or formally participate in SCEs; however, they may observe SCE teams while they conduct evaluations on site. These observation trips, lessons-learned reports, and experiences have been major contributors to SCE method's evolution into its current form. [BARBOUR93]

NOTE: ESC can provide SCE evaluation teams upon request for Air Force procurements. Contact ESC for more information *[see Volume 2, Appendix A, "Evaluating C3 Systems," for phone number and address].* **For more information on the SEI Transition Partners, contact the SEI** *[see Volume 2, Appendix A].*

CHAPTER 7 Software Development Maturity

MATURITY MODELS

Many organizations have embarked on organizational improvement efforts that focus on software process improvement. The **CMM[SM]** provides a means for assessing current practice and guiding process improvement efforts. *[The CMM[SM] is a registered Service Mark of Carnegie Mellon University.]* The International Standards Organization/International Electrotechnical Commission (ISO/IEC) has developed a version of the CMM[SM] framework, the **Software Process Improvement Capability dEtermination (SPICE)** model. There are also **specialty engineering CMM[SM]s**, based on the generalized engineering CMM[SM] framework. The models discussed here include:

- Capability Maturity Model (CMM[SM]) for Software,
- Software Process Improvement Capability dEtermination (SPICE),
- People — Capability Maturity Model (P-CMM[SM]),
- Software Acquisition — Capability Maturity Model (SA-CMM[SM]),
- Systems Security Engineering — Capability Maturity Model (SSE-CMM[SM]),
- Trusted Software — Capability Maturity Model (TS-CMM[SM]), and
- Systems Engineering — Capability Maturity Model (SE-CMM[SM]).

These models address many aspects of software development; however, most of the models (and the methods for using them) are not yet fully mature. *[For more information on the status of any of the CMM[SM]s, contact the SEI (see Volume 2, Appendix A).]* The CMM[SM]s contain three main components:

- **Key process areas (KPAs)** define the requirements that must be satisfied to accomplish each level of maturity. There are three categories of KPAs: management, engineering, and organization, as illustrated on Figure 7-1 (below). Because progress is made in stages or steps, the levels of maturity and their KPAs provide a road map for attaining higher levels. The KPAs at any given level describe the minimum requirements for that level. This does not mean some portion of those requirements cannot be satisfied or performed at a lower level, in fact they typically will. However, an organization cannot achieve the next higher level unless all KPA requirements have been satisfied. Once a higher maturity level is achieved, the models also make sure that all lower-level requirement satisfaction is maintained. The model stages are complimentary and flow upward. For example, the tracking and oversight at Level 2 will result in corrective actions (e.g., reactive approach to defects).

This process matures and compliments risk management at Level 3, where efforts are made to identify and prepare for risks before they happen (proactive approach). While defects should decrease as maturity increases, the need for corrective actions (established at Level 2) never completely goes away.

- **Interpreted KPAs**, developed for each general CMM[SM] KPA where additional interpretation, guidance, or practices are necessary to meet a particular discipline's needs. These are usually developed for engineering KPAs, since engineering disciplines typically have differences that must be addressed.
- **Specialty KPAs** address areas (other than management, engineering, or organization) specific to a particular discipline.

To facilitate the use of this family of models, general CMM[SM] KPAs are used to address common practices across all disciplines. Where needed, interpreted KPAs provide additional guidance on general CMM[SM] KPAs for a specific discipline. If additional new KPAs are required, specialty KPAs are developed. Specialty KPAs address those activities particular to a specialty discipline that go beyond general CMM[SM] KPAs. Interpreted KPAs and specialty KPAs are usually developed for general CMM[SM] engineering KPAs. In a similar manner, interpreted KPAs and specialty KPAs can be developed for management or organizational KPAs; however, this is rarely required. How the family of specialized CMM[SM]s relate to generalized CMM[SM]s (with the exception of SPICE and SE-CMM[SM]), is illustrated on Figure 7-1. [FERRAIOLO95]

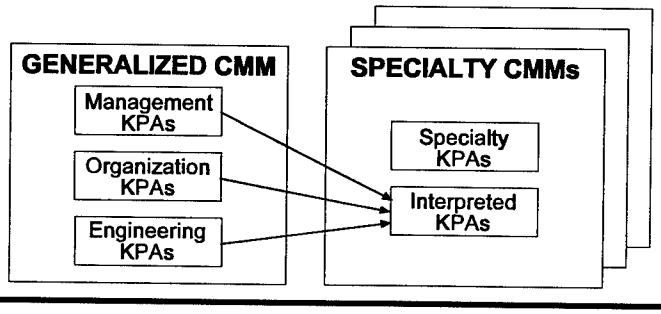


Figure 7-1 A Family of CMM[SM]s [FERRAIOLO95]

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on metrics and the CMM.[SM]

Capability Maturity Model (CMM[SM])

Like habitual and subconscious actions, software development processes are difficult to establish and even more difficult to break. Improvement seldom occurs by simply defining a more efficient process. Software engineers must understand the need to change, be convinced the new process will, indeed, improve performance, and be supported while they learn and implement it. The development processes for major software-intensive systems are often large and extremely complex. Therefore, they are difficult to define, comprehend, and especially, to implement. To aid organizations in determining the capabilities of their current process and to establish priorities for improvement, the SEI developed the **software process maturity framework**, as illustrated on Figure 7-2 (below).

The framework provides a benchmark of sound, proven principles for quality, recognized by both engineering and manufacturing disciplines to be effective for software. The purpose of the model is to help organizations determine their current capabilities and identify their most critical issues. The model characterizes the level of an organization's maturity based on the extent to which ***measurable*** and ***repeatable*** software engineering and management practices are institutionalized. This method can also be used to identify areas for improvement. Software managers usually know their problems in excruciating detail, but lack clear improvement priorities that can be understood and agreed upon by the team. By establishing a limited set of priorities and working aggressively to achieve them, more rapid progress can be made than with an unfocused effort. The CMM[SM] is organized into five maturity levels:

- **Level 1 — Initial**. The software process is characterized as *ad hoc*, and occasionally even chaotic. Few processes are defined and success depends on individual effort and heroics.
- **Level 2 — Repeatable**. Basic program management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on programs with similar applications.
- **Level 3 — Defined.** The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All programs use an approved, tailored version of the organization's standard software process for developing and maintaining software.

MATURITY LEVEL	CHARACTERISTICS	KEY CHALLENGES	RESULTS
5 OPTIMIZING	- Improvement fed back into the process - Automated tools used to identify weakest process elements - Numerical evidence used to apply technology to critical tasks - Rigorous defect-causal analysis and defect prevention	-Still human-intensive process -Maintain organization at optimizing level	PRODUCTIVITY / QUALITY / RISK
4 MANAGED	(Quantitative) - Measured process - Minimum set of quality and productivity measurements - Process data stored, analyzed, and maintained	- Changing technology - Problem analysis - Problem prevention	
3 DEFINED	(Qualitative) - Process defined and institutionalized - Software Engineering Process Group leads process improvement	- Process measurement - Process analysis - Quantitative quality plans	
2 REPEATABLE	(Intuitive) - Process dependent on individuals - Basic project controls established - Strength in doing similar work, but new challenges present major risk - Orderly framework for improvement lacking	- Training - Technical practices (reviews, testing - Process focus (standards, process groups)	
1 INITIAL	(Ad hoc/chaotic process) - No formal procedures, cost estimates, project plans - No management mechanism to ensure procedures are followed - Tools not well integrated; change control is lax - Senior management does not understand key issues	- Project management - Project planning - Configuration management - Software quality assurance	

Figure 7-2 Software Process Maturity Framework

- **Level 4 — Managed**. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
- **Level 5 — Optimizing**. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies. [HUMPHREY90]

NOTE: DoD's goal is to achieve a maturity Level 3 (*Defined Process*) for in-house Central Design Activities/Software Design Activities and weapon system Software Support Activities. Current practice encourages all bidders on software contracts to have a mature, well-defined, standardized process.

Except for Level 1, each maturity level is decomposed into several KPAs that indicate the areas on which an organization should focus to improve its software process. The KPAs at Level 2 focus on establishing basic program management controls. The KPAs at Level 3 address both program and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management across all programs. The KPAs at Level 4 focus on establishing a quantitative understanding of software process and work products under development. The KPAs at Level 5 cover issues that the organization and programs must address to implement continuous and measurable process improvement. Each KPA is described in terms of the key practices that contribute to satisfying its goals, and the infrastructure and activities contributing most to their effective implementation and institutionalization as the organization moves toward higher maturity levels.

ISO/IEC Maturity Standard: SPICE

International Standards Organization/International Electrotechnical Commission (ISO/IEC) is creating a set of international standards under the **Software Process Improvement Capability dEtermination (SPICE)** Program. One objective of the ISO/IEC effort is to create a process measuring capability, while avoiding any specific approach to improvement, such as CMM[SM] maturity levels. SPICE measures the implementation and institutionalization of specific processes. Organizations will be able to use this standard for:

- **Self-assessment** (to help determine an organization's ability to implement a new software program);
- **Process improvement** (to help an organization improve its own software development and maintenance processes); and
- **Capability determination** (to help a purchasing organization determine the capability of a potential software supplier).

SPICE Product Suite

The core set of SPICE products comprising the software process assessment standard include:

- **Introductory Guide.** This is the SPICE product suite entry point specifying what is required to obtain a SPICE-conformant assessment.
- **Baseline Practices Guide (BPG).** This identifies practices essential to good software management, engineering, and increasing process capability similar to the SEI's CMM.[SM]
- **Process Assessment Guide.** This describes the steps for performing an assessment and rating the organization against BPG practices.
- **Assessment Instrument.** This describes what type data to gather in an assessment and includes an example questionnaire of adequacy indicators for BPG practices.
- **Assessor Training and Qualification Guide.** This contains criteria for determining whether a candidate is qualified to perform a SPICE assessment.
- **Process Improvement Guide.** This guides an organization in applying other SPICE products for improving its software processes.
- **Process Capability Determination Guide.** This guides an organization in applying other SPICE products for selecting capable suppliers.

Baseline Practices Guide

The **Baseline Practices Guide (BPG)** defines, at a high level, the goals and fundamental activities essential to good software engineering practice. The BPG describes what activities are required — not how to implement them. BPG practices may be extended through Practice Guides that address a specific industry, sector, or other requirements. *[The CMM[SM] is an example of a sector-specific Practice Guide for large, software-intensive programs and organizations.]* The BPG defines five process categories:

- **Customer-supplier.** This category consists of processes that directly impact the customer, support development and transition of the software to the customer, and provide for its correct operation and use.
- **Engineering.** This category consists of processes to directly specify, implement, or maintain a system, a software product, and its user documentation.
- **Program.** This category consists of processes to establish the program and coordinate and manage its resources to produce customer satisfactory products or services.
- **Support.** This category consists of processes enabling and supporting performance of other program processes.
- **Organization.** This category consists of processes establishing organizational business goals and developing process, product, and resource assets to achieve business goals.

Each process in the BPG can be described in terms of **base practices** unique to software engineering or management activities. Process categories, processes, and base practices provide a grouping by type of activity. These processes and activities characterize performance of a process, even if it is not systematic. Performance of base practices may be *ad hoc*, unpredictable, inconsistent, poorly planned, and/or result in poor quality products, but those work products are, at least marginally, usable in achieving process purpose. Implementing only process base practices of a process may be of minimal value and represent only the first step in building a process capability. However, the base practices represent unique, functional process activities when instantiated in a particular environment.

BPG Capability Levels

The BPG expresses evolving process maturity in terms of capability levels, common features, and generic practices. A **capability level** is a set of common features (sets of activities) that, when applied together, increase a developer's ability to perform a process. Each level represents a major process capability improvement and process performance growth. They constitute a rational way for practice progression, harmonize different software processes rating approaches (i.e., the CMM[SM]). Capability levels provide two benefits: (1) they acknowledge dependencies among process practices; and (2) they help identify which improvements might be performed first, based on a plausible process implementation sequence. The BPG lists six capability levels:

- **Level 0 — Not performed.** This level has no common features and there is a general failure to perform base practices. There are no easily-identifiable process work products or outputs.
- **Level 1 — Performed informally.** Base practices are generally performed and process work products testify to performance.
- **Level 2 — Planned and tracked.** Process base practice performance is planned, tracked, and verified. Work products conform to specified standards and requirements. The primary distinction from the previous level is that process performance is planned, managed, and progressing towards being well-defined.
- **Level 3 — Well-defined.** Base practices are performed according to a well-defined process using approved, tailored versions of standard, documented processes. The primary distinction from the previous level is that the process is planned, managed, and standardized throughout the organization.
- **Level 4 — Quantitatively controlled.** Detailed measures of performance are collected and analyzed. This leads to a quantitative understanding of process capability and an improved ability to predict and manage performance. The quality of work products is quantitatively known. The primary distinction from the previous level is that the defined process is quantitatively understood and controlled.
- **Level 5 — Continuously improving.** Quantitative process effectiveness and performance efficiency goals (targets) are established based on organizational business goals. Continuous process improvement against these goals is enabled by quantitative feedback from defined process performance and the piloting of innovative ideas and technologies. The primary distinction from the previous level is that the defined, standardized process undergoes continuous refinement and improvement based on a quantitative understanding of the impact of process changes.

Common Features and Generic Practices

A **common feature** in the BPG is a set of practices (called **generic practices**) that address the aspects of process implementation and institutionalization. The words *"common"* and *"generic"* convey the idea that these features and practices are applicable to any process, with the goal of enhancing the capability to perform that process. For example, *"planning"* is a feature common to improved management of any process. Table 7-1 lists BPG common features and generic practices by capability level.

CHAPTER 7 Software Development Maturity

CAPABILITY LEVEL	COMMON FEATURE	GENERIC PRACTICE
LEVEL 5 Continuously Improving	Improving Organizational Capability	-Establish process effectiveness goals -Continuously improve the standard process
	Improving Process Effectiveness	-Perform causal analysis -Eliminate defect causes -Continuously improve the defined process
LEVEL 4 Quantitatively Controlled	Establishing Measurable Quality Goals	-Establish quality goals
	Objectively Managing Performance	-Determine process capability -Use process capability
LEVEL 3 Well-Defined	Defining a Standard Process	-Standardize the process -Tailor the standard process
	Performing the Defined Process	-Use a well-defined process -Perform peer reviews -Use well-defined data
LEVEL 2 Planned and Tracked	Planning Performance	-Allocate resources -Assign responsibilities -Document the process -Provide tools -Ensure training -Plan the process
	Disciplined Performance	-Use plans, standards, and procedures -Do configuration management
	Verifying Performance	-Verify process compliance -Audit work products
	Tracking Performance	-Track with measurement -Take corrective action
LEVEL 1 Performed Informally	Performing Base Practices	-Perform the process

Table 7-1 BPG Capability Levels, Common Features, and Generic Practices [KONRAD95]

BPG capability levels and CMM[SM] maturity levels are similar, yet distinctly different. BPG capability levels are applied on a per process basis, while CMM[SM] organizational maturity levels are a set of process profiles. Also, the BPG architecture does not prescribe any specific organizational improvement path. Improvement priorities are left completely up to the software organization, as determined by its business objectives. Individual processes, at either organization or program level, can be measured and rated using the BPG continuous

improvement architecture. [KONRAD95] *[For further information, contact the SEI (see Volume 2, Appendix A.)]*

People — Capability Maturity Model (P-CMM[SM])

Orangizations trying to improve their capability often discover a number of interrelated components must be addressed. Three necessary components for improvement are: people, process, and technology, as illustrated on Figure 7-3.

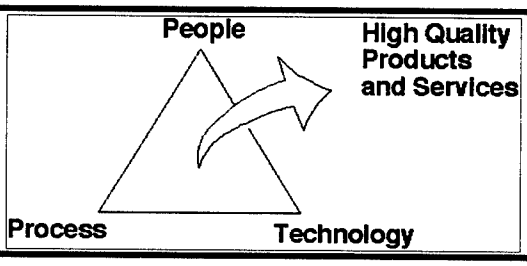


Figure 7-3 Three Necessary Components for Improvement [HEFLEY95]

Despite the importance of a talented staff, human resource practices are often *ad hoc* and inconsistent, and managers are insufficiently trained in performing them. Consequently, software managers often rely on their human resource departments for human resource practices administration (such as training, professional development, mentoring). Thus, these practices are applied with regard to how they impact performance. In many cases, even when software organizations are aware of the problem and want to improve these practices, they do not know where or how to begin.

The SEI's **People — Capability Maturity Model (P-CMM[SM])** provides guidance on how to improve human resource management. The P-CMM[SM] is an adaptation of the CMM[SM] that focuses on developing organizational talent. It can be used to radically improve an organization's ability to attract, develop, motivate, organize, and retain the talent needed to increase software development maturity. The P-CMM[SM] helps software organizations to:

- Characterize people management maturity,
- Set priorities for improving the level of talent,
- Integrate talent growth with process improvement, and

- Establish a culture of software engineering excellence that attracts and retains the best and the brightest.

P-CMM[SM] Structure

The P-CMM[SM] is fashioned after the CMM[SM] in structure and format. The P-CMM[SM] will evolve to stay synchronized with architectural changes made in the CMM[SM] and other maturity standards, such as SPICE. It provides the same type guidance as the CMM,[SM] but in a different dimension. **People management maturity** describes an organization's ability to consistently improve the knowledge and skills of its staff and align their performance with organizational objectives. The P-CMM[SM] addresses a broad range of people management issues, including:

- **Recruiting** (attracting talent),
- **Selection** (choosing talent),
- **Performance management** (coaching talent),
- **Training** (enhancing talent),
- **Compensation** and **reward** (rewarding talent),
- **Career development** (developing talent),
- **Organization** and **work design** (organizing talent), and
- **Team** and **culture development** (integrating talent).

As illustrated on Figure 7-4 (below), the P-CMM[SM] consists of five maturity levels. Each maturity level is a well-defined evolutionary plateau that institutionalizes a level of capability within the organization. Each level contains numerous KPAs designed to satisfy a set of goals set in the context of how people management practices are defined.

For instance, the KPAs at Level 2 focus on instilling basic discipline into people management activities. The KPAs at Level 3 address the issues of identifying primary competencies and aligning people management activities with them. The KPAs at Level 4 focus on quantitatively managing organizational growth in people management capabilities and in establishing competency-based teams. The KPAs at Level 5 cover continuous improvement methods for developing competency at the organizational and individual level. The KPAs are internally organized by common features (i.e., those attributes indicating whether KPA implementation and institutionalization is effective, repeatable, and lasting). The five common features are: commitment to perform, ability to perform, activities performed,

measurement and analysis, and verifying implementation. [HEFLEY95]

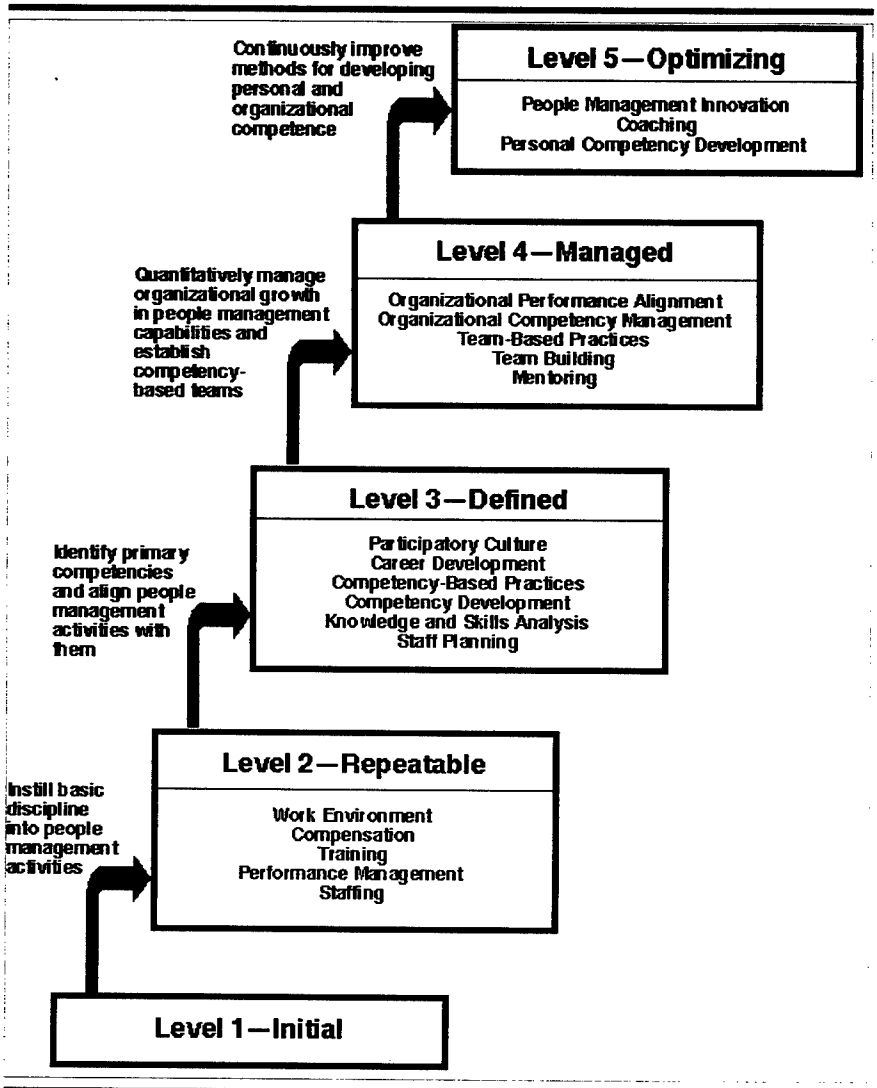


Figure 7-4 P-CMM[SM] Key Process Areas by Maturity Level [HEFLEY95]

Software Acquisition — Capability Maturity Model (SA-CMM[SM])

The SEI's **Software Acquisition — Capability Maturity Model (SA-CMM[SM])** was developed to assess the government's internal software acquisition management process maturity. It reflects a collaborative team effort by acquisition experts from DoD, federal agencies, the SEI, and industry, and provides a framework for benchmarking and improving the software acquisition process. Its users are those organizations with responsibility for acquiring and supporting software-intensive products, e.g., government PMs/PEOs, government Software Support Activities, industry PM/PEO equivalents, and senior executives. The purpose of the SA-CMM[SM] is to:

- Support senior management goal setting (i.e., each level of maturity represents an increased software acquisition process capability); and
- Support prediction of potential performance (includes accounting for factors significantly contributing to process capability).

The SA-CMM[SM] is based on the premise that, ***as we mature and improve our capabilities, our probability of success increases, and we are able to make better predictions***. The purpose of assessing an acquisition organization's maturity level is to identify areas for process improvement. To make improvements, an organization must have an ultimate goal, know what is required to achieve that goal, and be able to measure progress towards achieving it. The SA-CMM[SM] provides the information and guidance needed to facilitate those activities.

The SA-CMM[SM] defines KPAs for four of five maturity levels. While the SA-CMM[SM] describes the acquirer's role (in contrast to the CMM[SM] which focuses on the contractor's process), it includes certain precontract award activities, such as software Statement of Work preparation and documentation requirements, and source selection participation. The SA-CMM[SM] has the same architecture as the CMM,[SM] as illustrated in Figure 7-5 (below). SA-CMM[SM] maturity levels are described as:

- **Level 1 — Initial.** At Level 1, the program team typically does not provide a stable environment for acquiring software. Staffing is based on individual availability, resulting in a random composition

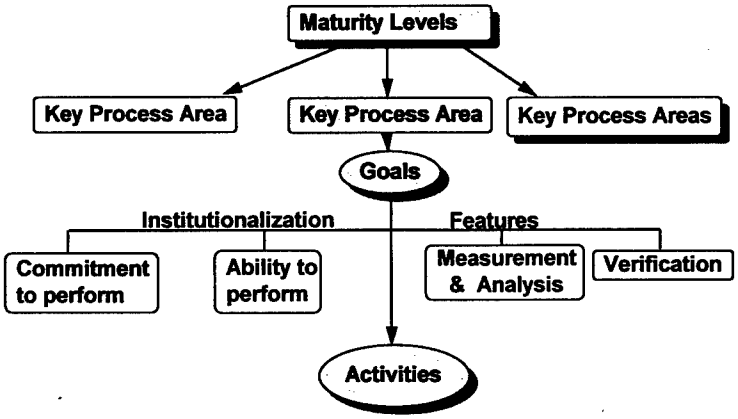


Figure 7-5 SA-CMM[SM] Architecture [FERGUSON95]

of acquisition skills. Normally, no one is well-versed in the specialized acquisition of software, nor is anyone dedicated to managing its acquisition. The program is conducted in an *ad hoc* manner.

- **Level 2 — Repeatable.** At Level 2, the program team is knowledgeable and supportive of promulgated policies, regulations, and standards relating to the software-intensive aspects of their program, and make a dedicated attempt to comply therewith. Software acquisition management plans and procedures are established. New program planning and tracking is based on experience with similar programs. An objective in achieving Level 2 is to stabilize contract management by allowing repetition of successful practices employed on prior programs. Contract planning and tracking is stable because documented oversight is in place.
- **Level 3 — Defined.** At Level 3, the organization's standard software acquisition process is defined and integrated into the program for both software contract and program management. The software acquisition process group facilitates process definition and improvement efforts. Processes established at Level 3 are used by management and staff and are tailored, as appropriate, for more effective performance. An organization-wide training program is implemented to ensure all practitioners and managers have the knowledge and skills required to carry out their tasks. A standard software acquisition process is well-defined and understood, and management has visibility into technical program progress. Compliance with plans and contract requirements is ensured, and the team works with the contractor to resolve compliance difficulties as they arise. Cost, schedule, and requirements are under control and software quality is tracked.

- **Level 4 — Managed**. Contracts have been implemented with well-defined, consistent measures establishing the quantitative foundation for evaluating program processes, products, and services. Control over processes, products, services, and contracts is achieved by narrowing performance variation to within acceptable quantitative boundaries using statistical process control. An organization-wide process database is used to collect and analyze defined software acquisition process data. Process and product quality are predictable and occur within quantitative, measurable limits. When these limits are violated, corrective action is taken.
- **Level 5 — Optimizing**. Focus is on continuous process improvement and identification of candidate processes for optimization. Statistical evidence is available to analyze process effectiveness and used to refine policies. Technological innovations exploiting best software acquisition management and engineering practices are identified, evaluated, and institutionalized. Level 5 organizations continuously strive to raise the upper bound of their process capability. Improvement occurs by incremental advancements in existing process and by innovations using new technologies and methods. [FERGUSON95]

Systems Security Engineering — Capability Maturity Model (SSE-CMM[SM])

The **Systems Security Engineering — Capability Maturity Model (SSE-CMM[SM]),** based on the CMM,[SM] establishes a metric for measuring security engineering maturity and provides a guide for improving organizational and production assurance and measurement techniques. SSE-CMM[SM] KPAs are illustrated in Figure 7-6 (below). [*NOTE: This information was taken from an April 1995 presentation at the Software Technology Conference. The SSE-CMM[SM] effort has been undergoing continuous improvement, and additional KPAs will be finalized after these Guidelines go to press. For the latest information on SSE-CMM,[SM] see their Web page [see "Industry" listing in Volume 2, Appendix B or call the point of contact listed in under "Government" in Volume 2, Appendix A.]* System security engineering KPAs focus on the development organization and the interface between customer roles and groups. The system security engineering KPAs are placed at specified maturity levels based on the process capability identified at each level. SE-CMM[SM] maturity levels include:

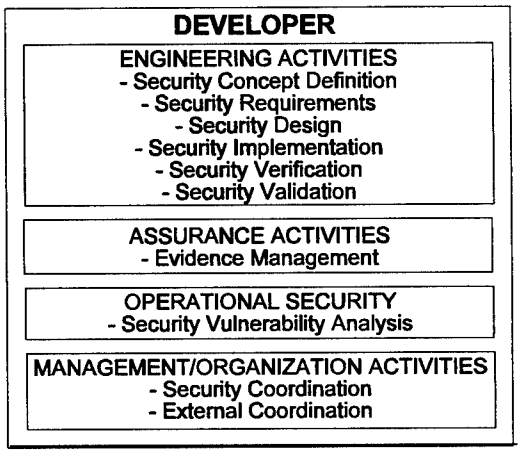


Figure 7-6 System Security Engineering CMM[SM] KPAs [FERRAIOLO95]

- **Level 1 — Initial.** At Level 1, few processes are defined and success depends on individual, heroic effort. While processes may be defined for some security engineering activities, there is no management mechanism to ensure they are performed. No KPAs are placed at this level.
- **Level 2 — Repeatable.** Institutionalization of effective security engineering processes allow successes on programs with similar applications to be repeated. Special security engineering KPAs at this level include:
  - Program management of security engineering aspects (program management and planning); and
  - Management of the security engineering activities necessary to build security into a system (security requirements, security design, and security implementation).
- **Level 3 — Defined.** All programs use an approved, tailored version of the organization's standard processes. Special security engineering KPAs at this level include:
  - **Process definition** addresses the documentation and standardization of security engineering processes;
  - **Peer reviews** monitor defects in results;
  - **Security verification** ensures that basic security engineering process work products are consistent with each other;
  - **Evidence management** ensures that evidence activities support customer assurance needs and are well-integrated with security engineering activities and other engineering disciplines;

- **Security coordination** includes management-oriented activities that ensure security engineering activity coordination;
- **Inter-group coordination** addresses security engineering activity coordination with other engineering disciplines; and
- **Security concept** and **external coordination** represents understanding of and interaction with all external organizations, such as the end-users, testers, or IV&V.

- **Level 3 — Managed.** The organization operates within specified limits and results are within those limits. Special security engineering KPAs at this level include:
  - **Security validation** represents activities that provide a better understanding of end-results conformance to the customer security needs.
  - **Quality management** provides an understanding of end-result conformance to previously specified quality goals.
  - **Security vulnerability analysis** includes activities that provide a better understanding of end result residual vulnerabilities. These KPAs measure conformance of end-results to desired results.
  - **Quantitative process management** provides an understanding and control of program's processes and conformance with organizational defined processes.
- **Level 5 — Optimizing.** Better process understanding allows identification of inefficient activities and effecting controlled changes which improve capability ranges (limits). Special security engineering KPAs at this level include:
  - **Defect prevention** supports the identification and prevention of security engineering product defects.
  - **Technology change management** supports identification of new security technologies (e.g., tools, methods) and evaluation of their effect on organizational security engineering processes.
  - **Process change management** addresses controlled, effective change to organizational security engineering processes based on input from defect prevention and technology change management. [FERRAIOLO95]

Trusted Software — Capability Maturity Model (TS-CMM[SM])

Software trust is the quantification and qualification of evidence that the development process used to create and modify software yields a product that exactly satisfies specified requirements and counters targeted threats. [KITSON95] Trusted software systems use a variety of integrity measures in support of security policy, for the development and application phases, allowing their use in processing sensitive or classified information. A set of *trust principles* derived from the **Trusted Software Development Methodology (TSDM)**, developed by the Strategic Defense Initiative Organization, is integrated into the CMM[SM] to produce the **Trusted Software — Capability Maturity Model (TS-CMM[SM])**. The National Security Agency (NSA) initiated the TS-CMM[SM] in collaboration with SEI to improve upon NSA's traditionally lengthy post-development evaluations and increase use of commercial information security software.

The SSE-CMM[SM] *[discussed above]* focuses on the entire *software life cycle* by ensuring proper security requirements are built into and maintained within the software and the software process. In contrast, the TS-CMM[SM] focuses on the *software development process* and ensures counter threats are built into the software by disallowing any defects that might permit unwanted access. A **Trusted — Software Capability Evaluation (T-SCE)** may be used during source selection and contract monitoring, and a trusted internal process improvement (T-IPI) program may be used for process improvement. [KITSON95]

Systems Engineering — Capability Maturity Model (SE-CMM[SM])

The SEI's **Systems Engineering — Capability Maturity Model (SE-CMM[SM])** expresses essential characteristics of the basic technical, management, and support processes for systems engineering, and provides guidance in applying process management and institutionalization principles to the systems engineering process. The SE-CMM[SM] architecture adopts that of the SPICE program's **Baseline Practices Guide (BPG)**.

SE-CMM[SM] Architecture

Similar to the BPG, the SE-CMM[SM] architecture separates actual domain process characteristics — systems engineering — from the practices related to managing those processes. It provides generic and domain specific-guidance for process management. A **base practice** is defined as an engineering or management practice that addresses the purpose of a particular process area. Base practices are contained in 17 process areas, divided into three process area categories: program, engineering, and organization. For example, the SE-CMM[SM] contains the engineering process area *"integrate system,"* the purpose of which is to ensure all system elements work together. One base practice in this process area is to develop detailed interface descriptions implied by the systems architecture. Base practices provide state-of-the-practice type guidance. Systems engineering functions are described in the base practices exhibited in the process.

Generic practices *[defined above]* are divided into **common features** *[also defined above]* which are contained in process capability levels. For example, in Level 2 (Planned and Tracked), the common feature *"planning performance"* contains practices to allocate adequate process resources, assign responsibilities for product development, and provide adequate tools to support the process. As illustrated in Figure 7-7 (below), the advantage of the SE-CMM[SM] architecture is principles upon which the CMM[SM] is based are abstracted and expressed in such a way that they can be used to assess any organization's processes — i.e., its generic practices. On the other hand, essential process characteristics from a particular domain are also clearly expressed — i.e., its base practices. This architecture, isolates both types of practices and looks at them separately. They are then merged back together to build and design processes. In this way, enterprise domain and process management needs are addressed and supported. [KUHN95]

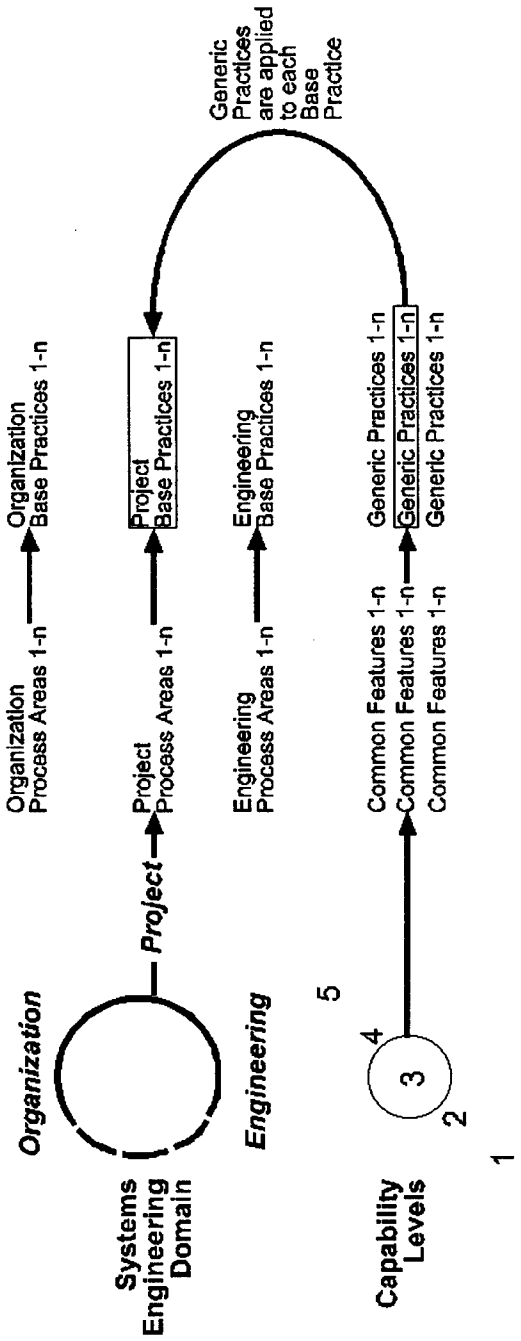


Figure 7-7 SE-CMM[SM] Architecture [BRIDGE94]

BENEFITS OF MOVING UP THE MATURITY SCALE

Published studies of software engineering improvements measured by the CMM[SM] indicate significant cost savings and return on investment (ROI). Thus, software testing and maintenance costs are decreased, because quality requirements are more readily met. Of the companies studied, a distinction is made between the one time CMM[SM]-compliance costs of achieving a higher maturity level from those costs for continuing to perform software engineering at that higher level. The latter has been found to actually represent a cost reduction when compared to software production costs at the former lower level. Some studies show that the one-time cost of achieving a higher level are quickly recouped by significant savings in producing higher-quality software, requiring less rework, that is easier to maintain.

All companies studied report that process improvement works best when employees and employer agree to accept the required extra effort and expense. One such arrangement is to have some meetings and training sessions conducted during the lunch hour, with the company providing lunch. Other variations on employer/employee compromises include *shared time*, when training is conducted on 50% company time and 50% employee time.

It is fairly easy to quantify the benefits of increased maturity at the company level. Production costs go down — quality goes up — time to market is shortened. How employees benefit is more subtle. The higher level in which an employee works, the more valuable he is to the software industry — i.e., the techniques learned are very marketable, useful professional skills. In addition, employee pride and management respect cannot be overlooked as an employee benefit, reward, and motivating force. Those companies having achieved higher maturity levels agree that a good reputation with their customers is primarily based on product quality and agreeable customer interrelations. Higher maturity levels lead to higher quality software, and therefore, increased company reputation. It also tends to change the manner in which companies interact with their customers. For example, the formality of a higher maturity level lessens *ad hoc* contractor tendencies to give into volatile government requirements, it also contributes to more reliable, mutually-beneficial contractor-government relationships. Above all, the most compelling benefit is also the most basic one: higher quality software, at lower cost, with improved company reputation, is a powerful formula for competing, winning, and keeping contracts. [SAIEDIAN95]

In the August 1994 report, ***Benefits of CMM℠-Based Software Process Improvement: Initial Results*** (CMU/SEI-94-TR-13), the SEI collected and analyzed data from 13 organizations (both industry and Government) to obtain process improvement results of efforts associated with the **CMM.℠** Table 7-2 summarizes these results. ***A 35% median productivity gain, a 19% schedule reduction, a 39% post-release defect reduction, and a 5:1 return on investment ratio bear convincing testimony to the worth of process improvement.*** The SEI stated that if these CMM℠ process improvements had been combined with more robust software engineering environments, the use of automated process control tools, or the implementation of methodology improvements [such as Cleanroom or peer inspections *(discussed in Chapter 15, Managing Process Improvement)*], the results would have been even more dramatic.

CATEGORY	RANGE	MEDIAN
Total yearly cost of SPI activities	$49,000 to $1,202,000	$245,000
Years engaged in SPI	1 to 9	3.5
Cost of SPI per software engineer	$490 to $2,004	$1,375
Productivity gain per year	9% to 67%	35%
Early detection gain per year (pre-test defects discovered)	6% to 25%	22%
Yearly reduction in time to market	15% to 23%	19%
Yearly reduction in post-release defect reports	10% to 94%	39%
Business value of investment in SPI (value returned on each dollar invested)	4.0:1 to 8.8:1	5.0

Table 7-2 Summary of SEI CMM℠ Software Process Improvement (SPI) Study

In this report, **quality** was defined as the state of software when released or delivered to customers. The most common measure of quality among the data submitted was the number of post-release field defect reports. Figure 7-8 illustrates yearly reductions in that number. The letter values on the Y-axis are arbitrary designations for organization anonymity. The number values in parentheses on the Y-axis indicate the number of years the organization invested in software process improvement (SPI) programs. Organization P sustained a

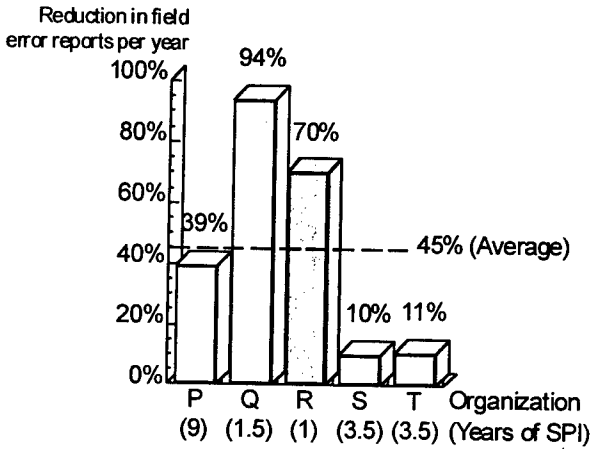


Figure 7-8 Reduction per Year in Post-release Defect Reports [SEI94]

remarkable defect report reduction rate of 39% per year over a 9-year period. That rate represents successive releases with substantial amounts of new and modified code — all of which completed its entire life cycle throughout that period. Organization P's last release had no defects reported in new and modified code. Organizations S and T also experienced substantial reductions for a significant period.

Productivity *[discussed on detail in Chapter 15, Managing Process Improvement]* data were gathered on lines-of-code (LOC) produced per unit of time. As illustrated on Figure 7-9 (below), the largest gain, organization G, was based on a comparison of two programs, only one of which adopted the SPI. The superior productivity of the second program was due to clear requirements definition and management. Organization H had a large productivity gain due to a *reuse [discussed in Chapter 9, Reuse]* program supported by tools and an environment adapted to promote reuse.

ROI data were reported in terms of the ratio of measured benefits to measured costs, as illustrated on Figure 7-10 (below). Benefits included savings from productivity gains and fewer defects. The benefits did not, however, include the value of enhanced competitive position from increased quality and shorter time to market. The cost of SPI included the cost of the SEPG, assessments, and training, but did not include indirect costs such as incidental staff time to put new procedures in place.

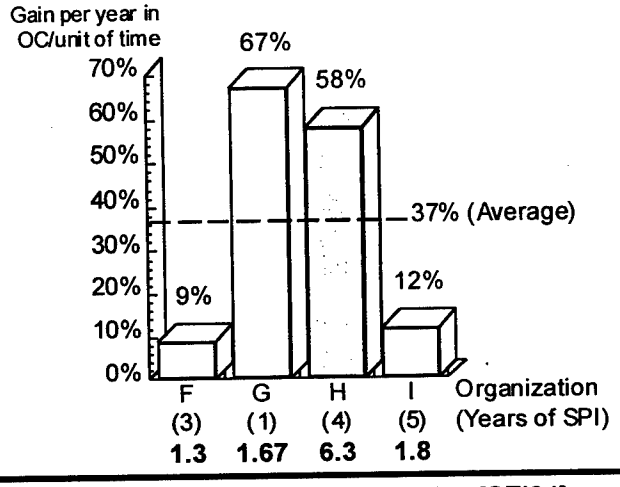


Figure 7-9 Gain per Year in Productivity [SEI94]

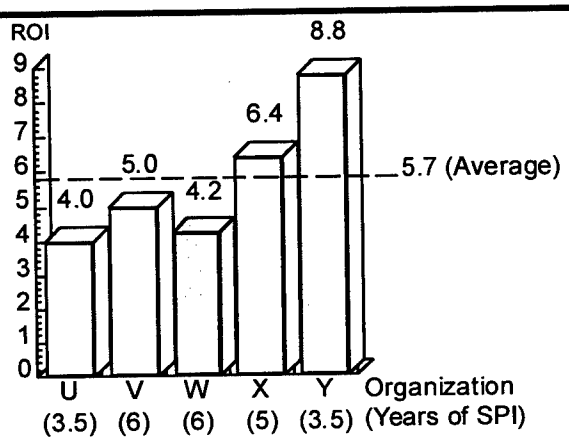


Figure 7-10 Return on Investment Ratio of SPI Efforts [SEI94]

Moving Up the Maturity Scale at USAISSDCL

The **US Army Information Systems Software Development Center Lee (USAISSDCL),** Fort Lee, Virginia, is one of three software development centers within the US Army, Information Systems Command (USAISC), headquartered at Fort Huachuca, Arizona. They are one of the largest software development centers in the Army with approximately 1,200 military, government civilians,

and contractor employees. Their systems encompass retail logistics, procurement, facilities engineering, food management, commissary, transportation, criminal investigation, and Army aircraft scheduling. Their operating budget is around $62 million with a customer base that spans the globe. On May 9, 1994, USAISSDCL was certified as a Level 3 organization, the result of an intense assessment conducted by SEI-authorized assessors under the auspices of the **Defense Information Systems Agency/Center for Information Management (DISA/CIM).**

Several factors contributed to their success, but chiefly, they worked hard and diligently towards achieving their Level 3. A major change in an organization's way of doing business is a touchy situation. For the USAISSDCL this was especially true, since the organization had been successful in achieving a Level 2 in 1990. It took them four years, however, of hard work, commitment, involvement, and patience to forge ahead to a Level 3. Commitment came from forward-thinking management who nursed the improvement program along during several downsizing initiatives. Management had the patience to allow the program to continue when immediate success was not evident. Management and the work force became educated in the change and improvement processes which made them aware that success takes time and often moves very slowly.

Inputs from analyzing their business processes, assessing prior Software Process Assessment (SPA) results, and evaluating other data gathering instruments, led to the development of a methodology for improvement. Their strategic improvement plan was derived from various sources, such as TQM, Functional Process Improvement (FPI), Business Process Re-engineering (BPR), CMM,[SM] their organizational vision and goals. All these inputs fed into the execution phase which incorporated customers' needs and converted those needs into activities. These activities were decomposed into processes and measured to show fundamental or continuous process improvement. The results of the execution phase were quality products, at a reduced cost, empowered teams, demonstrated ownership, and movement towards a Level 5 organization. [BOSWORTH95]

Moving Up the Maturity Scale at OC-ALC

The **Oklahoma City Air Logistics Center (OC-ALC),** Directorate of Aircraft (LA), Software Division (LAS), located at Tinker AFB, Oklahoma, employs over 400 software personnel in seven branches. In 1990 the SEI rated the LAS at a maturity Level 1 and by 1993 they were rated again at a Level 2, one of the first in the Air Force. Their goal is to achieve a Level 3 by 1996, the next scheduled SEI assessment. Since 1991, the CMM[SM] has been the basis for LAS process improvement efforts which involve developing and implementing a process improvement infrastructure. This infrastructure includes a Management Steering Team (MST) comprised of senior management, a Software Engineering Process Group (SEPG) comprised of technical people, and working groups for specific technical areas.

In 1993, an independent study was conducted to determine the economic benefits of software process improvement at LAS. Four LAS programs were studied, each of which were involved in the development of test program set (TPS) software to assess avionics circuit boards for three airplanes and one jet engine. The study covered a period of over eight years. The four programs selected were enough alike to make valid comparisons of their process improvement results. Table 7-3 summarizes the results of this independent study.

CATEGORY	RESULT
Return on investment	7.5 to 1 (An investment of $1.5 million resulted in a savings of $11.3 million)
Defect rates	90% reduction from the baseline program to the second program
Maintenance costs	26% reduction in average cost of a TPS maintenance action over the last 2 years
Productivity	10X increase from the baseline program to the most recent program

Table 7-3 OC-ALC Software Process Improvement Benefits [BUTLER95]

When three programs were compared to the baseline program their ROI was **7.5:1**. The study also showed that an investment of $1.5 million over an 8-year period resulted in cost savings of **$11.3** million. The $11.3 million figure was determined by factoring the amount the three programs would have cost had there not been any productivity improvements. Defect rates from the baseline program to the second program (the only other program for which there was

sufficient defect data) decreased by **90%** (3.30 defects/KSLOC compared to 0.28 defects/KSLOC for the second program). Data provided by one LAS customer showed that over the past two years, LAS reduced the cost of a TPS maintenance correction by **26%**. The most recent TPS development program studied was 10 times more productive than the baseline program (10 times more source code per manmonth was produced). This was attributed to improvements in process and technology from which the customer thought the program benefited from LAS process improvement efforts. [BUTLER95]

Moving Up the Maturity Scale at USSTRATCOM

The **US Strategic Command (USSTRATCOM)** is a joint service operation involving the Air Force, Navy, Army, and Marine Corps. The Command, Control, Communications, Computers, and Intelligence (C4I) directorate, known by its office symbol of J6, is responsible for USSTRATCOM's technical currency by providing software-intensive support necessary for war planning, command and control, intelligence, and office automation. The J6 has been challenged by increasing demands for fluid information systems and greater software productivity while under austere budgets with decreasing manpower.

In 1989, J6 (then part of the Strategic Air Command) was assessed at a maturity Level 1. An elaborate plan was developed outlining the various tasks the software production divisions needed to perform to reach maturity Level 2 by the self-imposed deadline of 1992. However, even though many engineers made a concerted, grass-roots effort to fulfill the taskings defined in that plan, the effort lost momentum within a year.

In 1991, process improvement efforts resurfaced in response to budgetary cutbacks. A postmortem of the previous effort revealed it failed because both accountability and management commitment were lacking. The 1989 plan placed the burden of improvement on the software divisions; yet they were neither held responsible for failing to reach improvement goals nor were they rewarded for maturing their software processes. The new approach took a total quality philosophy of forming an expert team responsible for garnering management commitment, marketing benefits to working-level personnel, and facilitating improvement throughout the organization. A three-phased approach of sponsorship, consultation, and training was implemented.

- **Phase 1 — Sponsorship.** Senior management created an internal software process improvement group to guide the efforts of the organization in achieving a maturity Level 2.
- **Phase 2 — Consultations.** The work of elevating the organization from Level 1 to Level 2 was accomplished through a series of reduced-scope CMM[SM] assessments, called *consultations*. The focus of the consultations was to have a group of expert software engineers assist individual divisions in interpreting, implementing, and performing Level 2 KPAs in their environment.
- **Phase 2 — Training.** The software process improvement group held periodic seminars to train personnel on the CMM[SM] and the benefits of process improvement. Seminars were conducted throughout the consultation process and gave J6 members an opportunity to learn about process maturity, to voice concerns, and to suggest changes to the improvement process.

Evidence of Improvement

The consultation visit to each software group revealed the number of key practices implemented by that group. Follow-up visits allowed the consultation team to track progress made in achieving previously implemented key practices. Indirect evidence of process improvement traceable to the consultation efforts included:

- The War Planning Systems Division had seven consecutive defect free database cutovers;
- Requirements processes in the War Planning Division were reduced from **14** to **2**;
- The Intelligence Systems Support Division, which had experienced an average of 250 defects during its 1989 software cutovers, reduced their defect rate to below **10** per cutover after the consultation effort;
- Command and control software cutover defects were reduced by **66%**, and labor costs were down **25%**;
- Command and control malfunction reports were reduced in two years from 219 to **19** per software release; and
- The Office Automation installation defect rates were reduced from 40% per installation to **3%** within 12 months.

The software maturation process has not only demonstrably improved J6's software business, but the by-products of the consultations has aided in performing long-range planning. One example was the implementation of a cost estimation effort for some 30 million lines-of-code within USSTRATCOM. Software engineers knowledgeable

with the COCOMO (Constructive Cost Model) *[discussed in Chapter 12, Strategic Planning]*, a model for estimating program costs as a function of software size and programming environment, worked with the software groups to assist in establishing an initial baseline for software costs. This baseline has proven accurate and has been an effective tool for forecasting the implications of customer work requests on cost, schedule, and performance.

Methods for Success

For J6, the process improvement effort has achieved its goal of rallying the software program groups into defining and managing their software processes. The specific methods used in the consultation process included:

- **Preventing defects**. The J6 maturation approach compelled software production groups to formalize the way they do business and to foster a ***proactive defect prevention*** *[discussed in Chapter 15, Managing Software Development]* culture and not a reactive defect-detection atmosphere.
- **Use of total quality principles**. Instead of viewing improvement efforts as a police action, the software groups accepted the improvement team of software experts as a strategic partner whose interests included the betterment of the entire organization.
- **Use of recognized tools.** The maturation process achieved a disciplined approach by employing tools, such as the CMM,[SM] process action teams, prepared questionnaires, interviews, follow-up reports, technology insertion, postmortem sessions. Information sharing, professional conferences, cost estimations, process documentation, brainstorming, program plans/schedules, and quality assurance training.
- **Use of systematic, integrated, and consistent standards.** The CMM[SM] provided the standard against which all software groups were evaluated. Every software group was appraised on its efforts in every Level 2 KPA.
- **Use of self-evaluation, feedback, and adaptation**. All software groups were appraised on one KPA before moving to the next KPA. At the end of each stage, the effort was reviewed and fine-tuned as necessary to improve the next cycle.
- **Basing results on quantifiable information**. Each consultation visit measured the number of practices being performed by the software group. Results were graphed and tracked for improvement.

- **Fostering continuous improvement.** The consultation approach was user-friendly and encouraged software production groups to embrace improvement efforts rather than coercing them into a mandatory program.

Process Maturity Pays Off

In March 1994, the improvement efforts of the USSTRATCOM were honored with the ***"Best of the Best"*** award from the Quality Assurance Institute of Orlando, Florida. That award recognizes information technology organizations that have demonstrated a superior and effective commitment to quality principles and practices. The Command won the award in the category of *"Best Unique/ Innovative Idea"* for its software development and maintenance activities. Previous improvement efforts had made little progress because they lacked management commitment and organizational focus. The use of a consultation process and process action teams gave the Command's software professionals the skills and innovation needed to improve their software processes and create a center of excellence. [UMPHRESS95]

Moving Up the Maturity Scale at SSG

In February 1993, a study report was published on the software engineering process at the **Standard Systems Group (SSC)**, Maxwell AFB-Gunter Annex, Alabama. The purpose of the study was to determine the economic benefits of:

- **Moving up the maturity scale** through software development process improvement; and,
- **Reuse of assets** from a library of standard reusable objects inherent in Integrated-Computer Aided Software Engineering (I-CASE) environment.

This study was based on two self assessments of the SSG to determine internal and overall maturity levels. Seven SSG programs were studied. Two of the seven advanced from a Level 1 to a Level 3, and three advanced to a Level 2, over the 5-year period between assessments. Differences were quantified between the original baseline and what was achieved when the program advanced. Comparisons were performed by analyzing the management metrics of schedule, effort [in person-months (PMs)], cost, peak manpower, and mean-time-to-defect (MTTD). Table 7-4 illustrates the changes in management metrics from a Level 1 to a Level 3 (for one program).

CHAPTER 7 Software Development Maturity

MANAGEMENT PARAMETERS	BEFORE SEI LEVEL 1 (1988)	AFTER (ACTUAL) SEI LEVEL 3 (1992)	DIFFERENCE BENEFIT	PERCENT DIFFERENCE	BENEFIT RATIO
Time (Months)	24.5	14.3	-10.2	-41.6%	1.71
Effort (PM)	1,494	263	-1,231	-82.4%	5.68
Uninflated Cost	$5,716	$1,008	($4,708)	-82.4%	5.67
Peak Staff (People)	100	31	-69	-69.0%	3.23
MTTD/Days	0.43	1.38	0.95	220.9	3.21

Table 7-4 Benefits of Moving from Level 1 to Level 3 (SSG Program Example)

Figure 7-11 illustrates the benefit ratios of selected management metrics (for the same program as Table 7-5) indicating the level of economic, productivity, and quality improvements. Without using the CMM[SM] management practices for a Level 2, one cannot move to a Level 3 or beyond, with merely the implementation of a software methodology or tool. Management processes are the foundation, not a software development methodology or tool. Picking tools without analysis of the software development methodology is the same as putting a methodology in place without the management practices required to carry an initiative through the entire product life cycle.

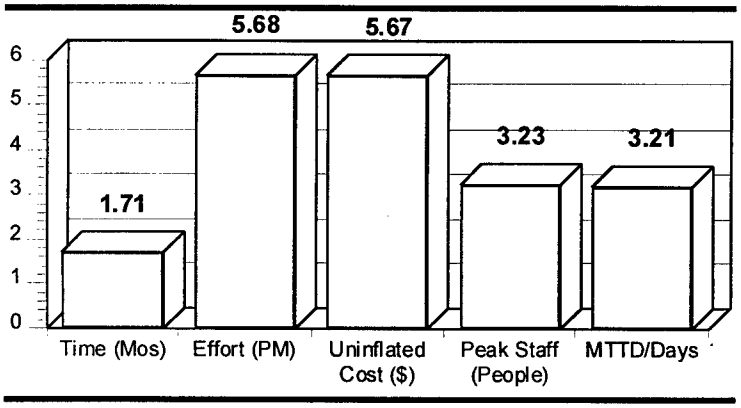


Figure 7-11 Benefit Ratios of Moving from Level 1 to Level 3

The report concluded that the data collected on the SSG programs indicate very significant benefits can be, and have been, attained through process improvement efforts. These include reduced schedule, staff, effort/cost, and defects resulting in much higher MTTD for fielded software. These savings, through increased productivity and quality, more than pay for the investment in equipment, software tools, and training required to achieve higher levels of software engineering maturity. [SSC93] This study also shows that there are substantive benefits to improving your software engineering capabilities. Implementing process improvement, reuse, quantitative planning, and measurement as part of your management process will produce a positive ROI as your development capability moves up the maturity scale. *[Reports Key Practices and the Capability Maturity Model and Capability Maturity Model for Software are available through DTIC and NTIS.]*

Moving Up the Maturity Scale at Raytheon's Equipment Division, Software Systems Lab

In 1988, the Software Systems Lab at Raytheon's Equipment Division performed an initial assessment based on the CMM[SM] questionnaire and found that they were slightly below a Level 2. They identified four areas needing improvement: documented practices and procedures, training, tools and methods, and metrics.

In 1992, a follow-up analysis of six major Raytheon Software Systems Lab programs over the three year period had substantially decreased rework costs. Since the start of the process improvement effort, Raytheon saved about **$9.2** million of its nearly $115 million software development costs. They quantified their software improvement cost savings by distinguishing the cost of doing something right the first time versus the cost of rework. This approach identified four major development cost categories:

- **Performance costs** associated with doing it right the first time (such as developing the design or generating code);
- **Nonconformance rework costs** (such as fixing code defects or design documentation);
- **Appraisal costs** associated with product testing to determine reliability; and
- **Prevention costs** in trying to prevent defects from degrading the product.

Nonconformance, appraisal, and prevention costs were defined as the **"cost of quality"** *[discussed in Chapter 15, Managing Process Improvement].* The primary objective was to significantly reduce nonconformance costs. This was accomplished, as evidenced by the **$9.2** million savings. Raytheon's numbers are quite remarkable. As summarized on Table 7-5, by investing almost $1 million annually in process improvements, Raytheon achieved a **7.7:1** ROI (a $4.48 million return on a $0.58 million investment) with **2:1** productivity gains. Raytheon says that it eliminated **$15.8** million in rework costs (from 41% to 11%) on 15 programs tracked between 1988 to 1992. Raytheon focused their process improvement efforts on three key areas.

CATEGORY	MEASUREMENT	RETURN ON INVESTMENT
Productivity	Increase in productivity	230% increase
Cost	Ratio of project $ saved to $ invested	7.80/1
	Project $ saved	$7.7M/year
	Rework	Reduced 75% ($17.2M savings since 1988)
Added benefits		Competitive position improved
		Higher employee morale
		Lower absenteeism/attrition rates
		Reduced overtime

Source: Raytheon (*IEEE Software*, July 1993); *Scientific American*, September 1994

Table 7-5 ROI at Raytheon by Moving from Level 1 to Level 3

- **Quantitative process management.** Raytheon's technical working group (TWG) created a Process Data Center to support proposal writing, quarterly reviews, software capability evaluations, and specific studies, such as the predictive mode necessary to achieve a Level 4 maturity. The TWG also provided standardized spreadsheet templates to facilitate metrics collection by program members.
- **Technology development.** Raytheon established a tools-and-methods working group that focused on evaluating tools and environments and process automation. The program sponsored the evaluation of alternative CASE products, cost-benefit analyses used to justify their purchase, training to instruct developers in their intricacies, integration of individual tools to provide a seamless

environment, tailoring of tools to specific programs, and generation of manuals for various types of users.

- **Training.** Raytheon sponsored a comprehensive training program with courses conducted during work hours (564 courses in 1992). Overview courses were designed to periodically provide general knowledge about specialized technical or management areas. Detailed courses were tailored for specific programs and scheduled accordingly. [SAIEDIAN95]

Moving Up the Maturity Scale at Hughes Aircraft Company, Software Engineering Division

In 1987, Hughes Software Engineering Division was assessed at a CMM[SM] Level 2. The SEI assessment team assessed six Hughes programs and identified seven areas needing process improvement: quantitative process management, process group, requirements, quality assurance, training, review process, and working relationships. In early 1988, Hughes developed an action plan to implement the recommended improvements. As summarized on Table 7-6, their 2-year program to raise the Software Engineering Division from a Level 2 to a Level 3 cost the company roughly $400,000 (75 person-months), a 2% increase in division overhead.

CATEGORY	MEASUREMENT	RETURN ON INVESTMENT
Cost	Ratio of project $ saved to $ invested	$445,000/$4M = 9/1
	Project $ saved	$2M/year
	Actual cost over budgeted cost	Decreased 50%
Schedule	Overruns	Decreased 50%
Added benefits		Increased management commitment
		Increased corporate pride
		Reduced overtime
		Reduced daily crisis management
		Created a more stable work environment

Source: SEI and Hughes (*IEEE Software*, July 1991)

Table 7-6 ROI at Hughes by Moving from Level 2 to Level 3

After implementing its process improvement program for two years, Hughes requested a second assessment by the SEI, which was performed in 1990. The company had progressed to a strong Level 3 with many activities preparing it for Levels 4 and 5. Hughes calculated that its initial ROI amounted to $2 million annually, based on a 50% reduction in its cost-performance index (CPI) (budgeted cost of work performed/actual cost). The ROI of this investment was **4.5:1**. Hughes' CPI continued to improve through 1992, climbing from 0.97 to **1.02**, to the point where, as a whole, their programs were under budget. Hughes attributes these savings to early detection of defects which substantially reduced rework costs. Hughes focused their process improvement efforts on three key areas.

- **Quantitative process management.** Hughes standardized uniform data definitions across programs and used them to track cost estimates, actual costs, defects, and schedule performance. A monthly report was compiled for senior management that included the program's accomplishments, problems, program trouble reports, quality indicators, scope changes, resource needs, and lessons-learned. Actual versus planned values were plotted over time to show the schedule, milestones, rate chart, earned-value, financial/ labor status, and target-system resource use.
- **Technology development.** Hughes' technology steering committee defined technology management practices and procedures and created a position called, the *"Head of Technology Transfer."* This individual monitored process maturity, maintained a technology database for each program, identified what technology each program needed, and was involved in corporate-wide technology development, process maturity, and training programs.
- **Training.** Hughes made training a job requirement rather than a promotional requirement. It supplemented its classes on programming practices, languages, and CASE tools with classes on program management, internal reviews, requirements writing, requirements and unit-level testing, and quality assurance. [SAIEDIAN95]

Moving Up the Maturity Scale at Litton Data Systems

In June of 1994, Litton Data Systems performed a licensed Software Process Assessment (SPA) of its software engineering process. The assessment team concluded that Data Systems had fully satisfied all of the goals for the Level 2 and Level 3 KPAs, with one exception: the Level 3 training goals were only partially satisfied. The reason they failed to achieve Level 3 was the assessment team found their training program was not planned in accordance with the Division's objectives and operations, as illustrated in Figure 7-12.

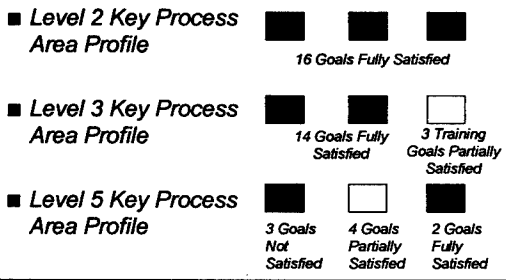


Figure 7-12 1994 Assessment Results [DIXON95]

After that assessment, a Division Software Training Plan was implemented to develop, fund, schedule, and provide the training necessary to meet the Division's current and future needs. This plan described the training of software personnel based on their job functions. It included the course descriptions, methods, schedule, update procedures, funding, and training program administration. It also contained provisions for verifying that training was effective and conducted according to the plan. Litton learned many lessons while implementing their Software Process Improvement Action Plan to achieve a Level 3, which included:

- **Management support**. The management at Litton was generous in providing adequate financial support for the effort necessary to form and support the SEPG and working groups.
- **Division-wide approach**. Working groups were staffed with over 60 experts from SQA, engineering program management, program office, system test, system engineering, engineering program management, software configuration management, business development, and representatives from the major programs.

- **Management oversight**. Litton's senior management took a genuine interest in the accomplishments of the SEPG. A Steering Committee reported weekly to the Director of Software Processes, bi-monthly to the Engineering Vice President and engineering directors, and bi-annually to the senior staff.
- **Functional teams**. An integral part of Litton's software process improvement effort involved transitioning to a functional team approach to software development which created an atmosphere that fostered communications among team members, other organizations within Litton, and customers. The team approach improved the predictability and efficiency of the software development process.
- **Communications**. The SEPG published a quarterly software engineering newsletter to share process and program news with the Litton's staff and customers.
- **Implementing change**. Litton used several means to institutionalize new procedures on on-going programs: (1) Software training was developed for the new processes; (2) a SEPG representative was assigned to each product area to serve as a consultant and to facilitate the institutionalization of the new processes; and (3) software program leaders attended Steering Committee meetings on a monthly basis to address program-specific process issues.
- **Institutionalize inspections**. Their formal peer inspection process was successful because of active management support, extensive inspection training, and because inspections were cost effective.
- **Structured working group activities**. Litton required that each working group develop a charter, a Process Improvement Plan, and a schedule. Working groups regularly compared their activities and accomplishments against the Action Plan and the CMM.
- **SEPG turnover**. Formerly, the SEPG and working groups suffered heavy attrition which was detrimental to working group momentum and resulting in wasted efforts, unnecessary rework, and slipped schedules. Litton, thus, assigned individuals who had contributed to software process improvement to other programs where they became champions of software improvement and supported the institutionalization of new processes.
- **Limited procedure reviewers and checklist use**. The review process is useful for process buy-in, but Litton limited reviewers to reduce confusion, hundreds of comments, and slipped schedules. Checklists were used to provide different instructions

for each reviewer, so that they would not all review a document for the same thing. [DIXON95] Figure 7-13 illustrates the characteristics of a Level 3 maturity.

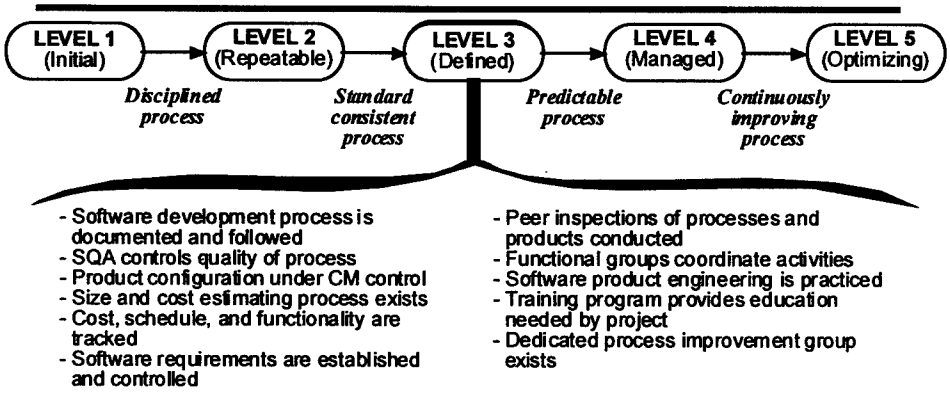


Figure 7-13 Characteristics of a Level 3 Contractor

ADDRESSING MATURITY IN THE RFP

To ensure the software process enacted for your program is predictable, repeatable, and manageable in terms of quality, cost, schedule, and performance, you should evaluate the offeror's software development capabilities prior to (or during) source selection. Remember, ***you are buying the process as well as the product!*** Performing a software development capability assessment will help you identify risks associated with the offeror's approach. Risk identification is possible, since you will have: (1) an understanding of how the organization managed software development efforts in the past; and (2) you will be able to compare past performance with the proposed software development process.

Therefore, you must pay due attention to the offeror's software development processes, starting with overall assessments like the **SCE** or **SDCE**, which focus on the details of tools, metrics, personnel facilities, management control, and language experience. Based on the maturity level of the selected contractor, you should consider customizing your contract to adapt that offeror's strengths and weaknesses. For example, if the contractor has achieved a high level of maturity (3 or above), you may decide that online access to the contractor's development environment and management status reports

(e.g., cost, schedule, risk management and metrics data) is an effective alternative to the traditional oversight mechanisms of formal reviews and submission/approval of data items. Alternatively, if an offeror's process for coordinating the efforts of different engineering disciplines and stake holders is relatively weak, you may add a requirement for an on-site liaison to support coordination with users and the contractors developing interfacing systems.

REFERENCES

[AUGUSTINE86] Augustine, Norman R., Augustine's Laws, Viking Penguin Inc., New York, 1986

[BOSWORTH95] Bosworth, Lawrence E., "The Process Improvement Highway," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[BRIDGE94] "Building Better Bridges: SEI Contributes to Development of a Systems Engineering Capability Model," Issue Four, *Bridge*, 1994

[BUTLER95] Butler, Kelley L., "The Economic Benefits of Software Process Improvement," *CrossTalk*, July 1995

[CLOUGH92] Clough, Anne J., "Software Process Technology," *CrossTalk*, June/July 1992

[DIXON95] Dixon, Susan A., "Litton Data Systems Attains Level 3 Software Process Maturity,"

[DSMC90] Defense Systems Management College, Systems Engineering Management Guide, US Government Printing Office, Washington, RUN-TIME, 1990

[FERGUSON95] Ferguson, Jack R., and Peter A. Kind, "A Software Acquisition Maturity Model," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[FERRAIOLO95] Ferraiolo, Karen M., and Joel E. Sachs, "Security and Security Engineering Overview: The Security Engineering CMM Effort," tutorial presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[HEFLEY95] Hefley, William E., et al., "People Capability Maturity Model (P-CMM) Incorporating Human Resources into Process Improvement Programs," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995

[HOROWITZ95] Horowitz, Barry M., personal communication to Lloyd K. Mosemann, II, December 1995

[HUMPREY95] Humphrey, Watts S., A Discipline for Software Engineering, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995

[KITSON95] Kitson, David H., "A Tailoring of the CMM for the Trusted Software Domain," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[KONRAD95] Konrad, Michael D., and Mark C. Paulk, "An Overview of SPICE's Model for Process Management," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995

[KUHN95] Kuhn, Dorothy A., and Suzanne M. Garcia, "Developing a Capability Maturity Model for Systems Engineering," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1995

[PATTON47] Patton, GEN George S., Jr., War As I Knew It, Houghton Mifflin Company, Boston, 1947

[SAIEDIAN95] Saiedian, Hussein, and Richard Kitzara, "SEI Capability Maturity Model's Impact on Contractors," *IEEE*, January 1995

[SEI94] *Benefits of CMM-Based Software Process Improvement: Initial Results* (CMU/SEI-94-TR-13), Software Engineering Institute, Carnegie-Mellon University, August 1994

[UMPHRESS95] Umphress, David A., Victor M. Helbling, John R. Russell, and Charles A. Keene, "Software Process Maturation: A Case Study," *Information Systems Management*, Spring 1995

CHAPTER 7
Addendum A

A Correlation Study of the CMM[SM] and Software Development Performance

Dr. Patricia K. Lawlis
Captain Robert M. Flowe (USAF)
Captain James B. Thordahl (USAF)

The Software Engineering Institute's (SEI) **Capability Maturity Model (CMM[SM])** is widely used to measure an organization's software development process maturity. The Department of Defense (DoD) has adopted this model with the belief that a more mature software development process will result in a more successful software program. Although there is a growing body of correlation, we were unable to find any studies, that validate this premise.

The study reported in this article — master's thesis research at the Air Force Institute of Technology (AFIT) — set out to find empirical evidence to determine if the premise could be validated. The goal of our research was to determine the nature of the correlation, if any, between software process maturity and software program success. We measured process maturity using a CMM[SM] rating, and success was measured using cost and schedule indicators.

First, we provide an introduction to the concepts and definitions we used in our work. Then, we provide details of our methodology, that describe our approach. Finally, we provide the details of our analysis, then summarize our results.

INTRODUCTION

The SEI's CMM[SM] has been widely accepted as a significant step toward solving the problems that plague the development of DoD software. By applying the process maturity assessment protocols to a potential software developer, the Government obtains an assessment of the developer's capability to produce quality software. Procurement risk is thus thought to be reduced, and the probability of obtaining the desired software within the constraints of schedule and budget are thought to be increased. The key assumption is that there is a significant positive correlation between the SEI CMM[SM] rating and the success of the software development.

Software Success

Although the concept of "*success*" in terms of software development has many definitions, a consistent theme found in software engineering literature is that success can be measured as a combination of cost, schedule, and quality performance [5,6]. Cost and schedule data have a variety of positive attributes that make it convenient for this type of correlational study. These data are readily available for a broad range of programs over a relatively long historical period and are in a relatively consistent format. Quality data, on the other hand, are reported sporadically at best with no accepted standard metrics let alone a stand form for reporting the measurements derived from such metrics. Thus, for the purposes of our research, success was defined cost and schedule performance.

Measuring Cost and Schedule

To measure cost and schedule performance, two steps must be followed:

1. A performance baseline must be established.
2. Actual performance must be compared to the baseline.

In program management, the Cost Performance Index (CPI) and the Schedule Performance Index (SPI) are the standard indices in use. Actual Cost of Work Performed (ACWP) is the sum of funds actually expended in the accomplishment of the planned work tasks. The Budgeted Cost of Work Performed (BCWP) represents the earned-value of the work performed and is an estimate of the value of the work completed. Deviations in the actual versus planned cost can be

expressed in the ratio of BCWP to ACWP. This is called the Cost Performance Index (CPI).

$$CPI = BCWP/ACWP$$

A CPI of less than 1.00 implies that for every dollar of value earned, more than one dollar was actually spent — a cost overrun. A CPI of more than 1.00 implies that for every dollar of value earned, less than one dollar was spent, and a CPI of 1.00 implies an "*on-target*" condition. Similarly, the programmed rate of funds expenditure is the Budgeted Cost of Work Scheduled (BCWS), which can be expressed as the planned expenditure of funds over time, based on the completion of the planned work packages. The ratio of BCWP to BCWS defines the degree to which a program is ahead of or behind schedule and is called the Schedule Performance Index (SPI).

$$SPI = BCWP/BCWS$$

An SPI of less than 1.00 implies that for every dollar of work scheduled, less than one dollar has been earned — a schedule overrun. An SPI of more than 1.00 implies that for each dollar of work scheduled, more than one dollar of work has been earned, and an SPI of 1.00 implies an on-target condition. [NICHOLAS90]

The indices of CPI and SPI are the standard cost and schedule performance measures for both Government and industry. The closer the CPI and SPI are to a value of 1.00, the more successful the program can be considered, at least in terms of cost and schedule. [NICHOLAS90] This establishes our performance baseline against which we can compare actual performance data.

The Effect of Process Maturity on Performance

Most of the CMM[SM] literature describes a positive relationship between process maturity and performance. As an organization matures from Level 1 to Level 5, the difference between target results and actual results decreases, i.e., CPI and SPI move closer to 1.00, and the variability of the actual results about the target decreases, i.e., performance becomes more predictable.

Graphically, the relationship between maturity and performance can be thought of as a probability distribution, see Figure 7-14 (below). At Level 1, the central tendency is somewhere below the target, and

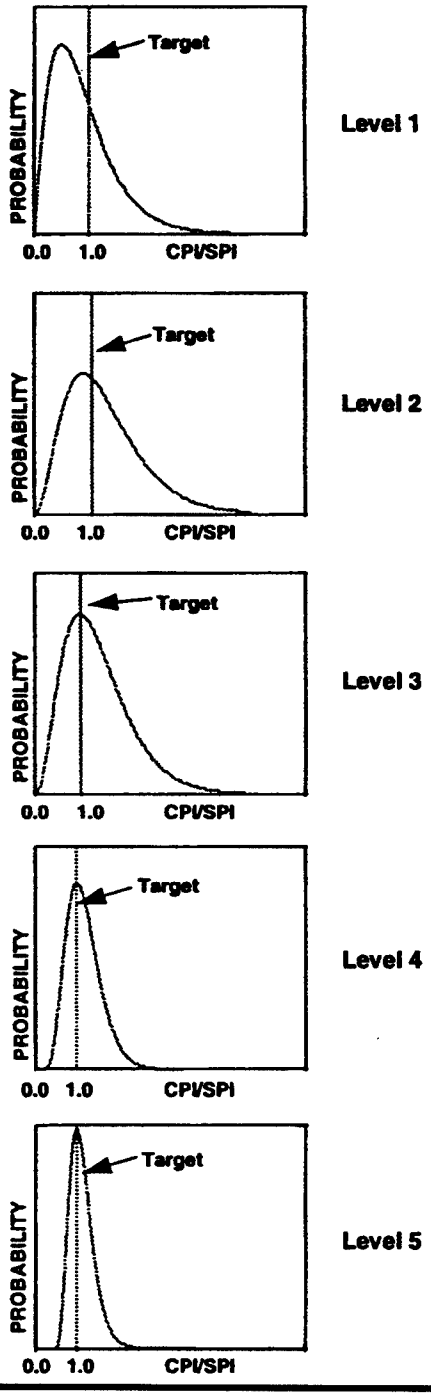


Figure 7-14 The Effect of Process Maturity on Performance

the distribution exhibits a high variance. At Level 2, the central tendency of the distribution is now on or very near the target, but the distribution still exhibits a high degree of variance. At Level 3 and above, the central tendency of the distribution is the same as the target, and the variance of the distribution decreases as the CMM[SM] rating increases. [PAULK93] This relationship between process maturity and performance is what we expected to see with respect to both cost and schedule performance. We had to set up a methodology to test this hypothesis scientifically.

METHODOLOGY

In order to examine the correlation between process maturity and program success, we collected historic data from Air Force software development contracts. We collected the data using a combination of archival search of cost libraries at target product centers, telephone inquiries, and personal interviews. Data were gathered in such a way as to preserve the anonymity of the program, the contractor, and all personnel involved. Target product centers were Aeronautical Systems Center (ASC) at Wright-Patterson Air Force Base, Ohio, and Electronic Systems Center (ESC) at Hanscom Air Force Base, Massachusetts. We focused on large programs that are required to track cost and schedule data according to the Cost/Schedule Control Systems Criteria (C/SCSC) guidelines and archive these data in the Cost Libraries at ASC and ESC. We limited our study to those programs developed for the Air Force which met the following criteria:

- Programs tracked software-specific cost and schedule data in C/SCSC format.
- Contractors were rated according to the SEI CMM[SM] protocols.
- Relevance of cost and schedule data to the rating could be established.

Many of the organizations providing data to the ASC and ESC cost libraries had been rated by the CMM[SM] protocols. We obtained the rating information through interviews with the program office personnel. It should be noted that we did not attempt to independently verify rating data. We established the relevance of the data by looking at what we called the temporal and associative aspects of relevance (see the next section). Those programs for which we could establish both a temporal and an associative relevance with a CMM[SM] rating established the dataset upon which we drew our conclusions.

Temporal and Associative Relevance

The degree to which the performance data are representative of the rating is important for the correlational analysis to be valid. Performance data and rating data must be linked by time (temporal relevance) and by association (associative relevance). Temporal relevance is obtained by collecting cost and schedule performance data over the 12-month period surrounding the rating date. Associative relevance depends upon whether the program under consideration was used in the CMM[SM] rating process. Four scenarios define the four degrees of rating-to-program relevance:

- **Very high relevance**. The program under consideration was the sole program evaluated in the CMM[SM] rating process.
- **High relevance.** The program under consideration was one program of several used in obtaining the CMM[SM] rating for the organization.
- **Medium relevance.** The program under consideration was not used to establish the CMM[SM] rating, but the organization or personnel who participated in the program were also responsible for programs evaluated in the CMM[SM] rating of interest.
- **Low relevance.** Neither the program nor the personnel responsible for the program under consideration were used to obtain the organization's CMM[SM] rating; the rating for the contractor as a whole is considered to apply to the organization responsible for the program under consideration.

We recognized that programs with medium and low rating-to-program relevance may adversely affect the validity of the correlation between rating and performance. At the outset, our concern for the scarcity of data militated against eliminating medium- and low-relevance programs from consideration. Instead, characterizing the relevance of the data enabled sample stratification, which permitted us to account for any relevance-related effects. Of the 52 data points evaluated, 40 had high to very high rating-to-program relevance.

Data Collection

We collected data on 11 DoD contractors who had been rated by the CMM[SM] protocols on 31 software programs these organizations were developing while their ratings were in effect. The net result of our data collection was 52 data points. In addition to rating and cost and performance index information, each data point was characterized by

moderating variables that were used to provide insight into the correlation between the performance indices and ratings.

For the purposes of this discussion, a data point is defined as an instance or set of circumstances where for a given software development program, rating data and cost data exist and are mutually relevant. Based on this definition, multiple data points may arise from a particular organization, program, or program. An individual organization may have multiple programs that fall within our sampling criteria. Additionally, each program may have one or more tasks that meet the sampling criteria, which means that the cost and schedule data were reported for individual software-unique work packages or tasks. Finally, each individual program may have been in progress during multiple rating periods, so cost and schedule data would be considered for each rating period.

Figure 7-15 illustrates an example for an organization (DoD contractor) that has been rated twice and has two programs (government contracts A and B), one of which has three individual software tasks (WBS

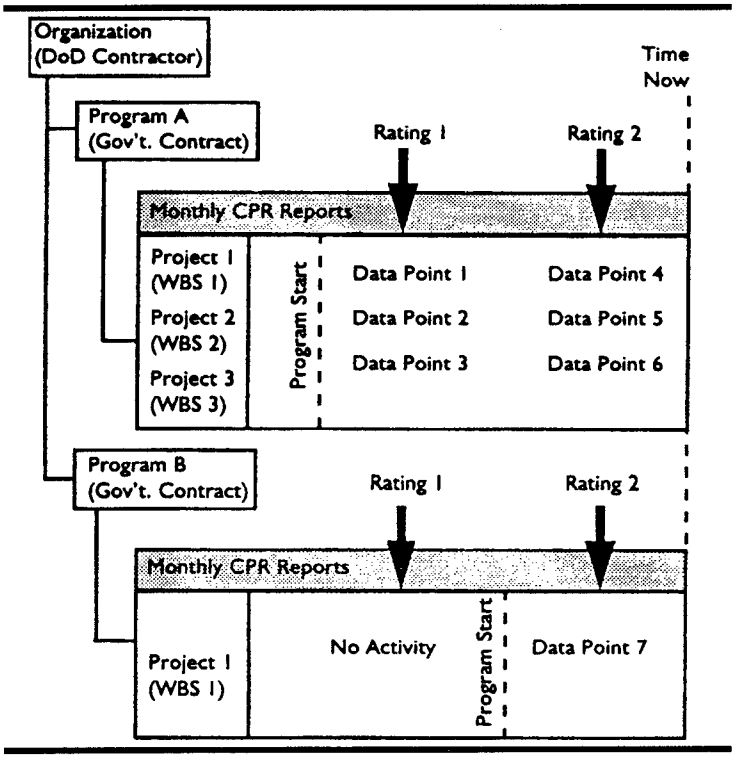


Figure 7-15 Origin of Data Points

elements), the other only one. Note that these government contracts have different periods of performance. Tasks 1 through 3 of Program A were in effect for two rating periods, whereas Tasks 1 of Program B was in effect for the last rating period only. In this scenario, this one organization would have provided seven discrete data points.

Table 7-7 summarizes the characteristics of the complete dataset used in this research. Each data point is also characterized by parameters that lend context to the data point. These parameters are called moderating variables and may provide insight into the factors that influence the correlation between the performance indices and the ratings. However, these parameters are not considered here at length. For a complete analysis, which includes these moderating variables, see the complete AFIT masters thesis. [FLOWE94]

CRITERION	DATASET COUNT
Number of contractors	11
Number of programs (contracts)	13
Number of programs (WBS elements)	31
Number of data points	52
Number of data points from ESC	35
Number of data points from ASC	17
Average number of data points per program	4.0
Average number of data points per program	1.7
Average number of ratings per contractor	1.9

Table 7-7 Characteristics of the Complete Dataset

It is important to note that all of the data points had CMM[SM] ratings of 1, 2, and 3. It is still highly unusual to find organizations with higher ratings, so these represent the expected data values. However, it means that our research findings are only valid for these three CMM[SM] levels.

Analysis

CMM[SM] rating data are at best ordinal in nature. Hence, statistical analysis techniques such as multiple linear regression, which require interval or ratio data, cannot be rigorously applied. However, a combination of descriptive and nonparametric techniques are adequate to establish the presence or absence of a statistically significant

to establish the presence or absence of a statistically significant correlation of software process maturity and software development success.

Our analysis focused on graphically and statistically correlating the cost and schedule performance indices with the respective CMM[SM] ratings. Graphical analysis tools included scatter plots and box and whisker plots. Statistical tools included Kruskal-Wallis nonparametric analysis of variance as well as the multiple comparison of mean rank test. It is not our intent to explain these tools at length, but we have provided a brief summary of the use of each in the next two sections. For complete discussions of these analysis tools, see the references provided below.

Graphical Tools

Scatter plots are used to visualize data by plotting two elements of each data point. Box and whiskers plots show similar data except that they indicate data groupings by a box and outliers by "*whiskers*," rather than plotting each individual point. [DEVORE82] For our analysis we used plots of CPI versus CMM[SM] rating as well as SPI versus CMM[SM] rating. See Figure 7-16 and 7-17 for scatter plots indicative of our analysis.

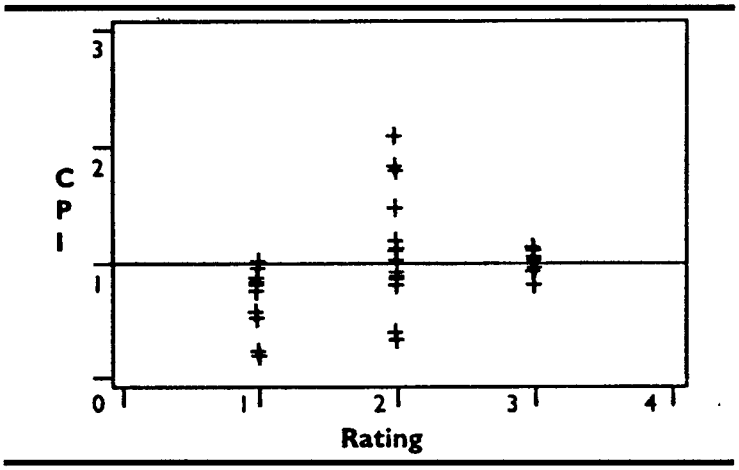


Figure 7-16 Scatter Plot of CPI versus Rating for High and Very High Rating Relevance

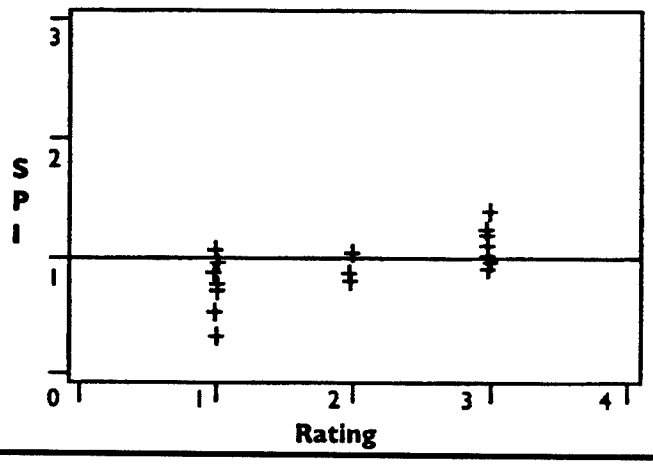


Figure 7-17 Scatter Plot of SPIU versus Rating for Less Than 80% Complete

Statistical Tools

The Kruskal-Wallis Test is a nonparametric analysis of variance, testing the null hypothesis that samples subjected to different treatments, e.g., CMM[SM] ratings, actually belong to the same population. The rejection of the null hypothesis suggests that the populations are different. [GIBBONS76] In our analysis, the rejection of the null hypothesis suggests there is a difference in the median performance of organizations at different maturity levels.

The Kruskal-Wallis test can only determine if at least two of the samples are from different distributions. To determine if there is a statistically significant difference in more than one pair of samples and which samples differ from which others, a multiple comparison test is required. The multiple comparison test compares the absolute values of the differences of the mean ranks between two samples to determine if there is a significant difference in the two samples. It is important to note that the overall level of significance used in multiple comparisons frequently is larger than those ordinarily used in an inference involving a single comparison. Our multiple comparison test was performed at a level of significance of 0.2, which implies an 80% level of confidence in the result. This is consistent with recommended values for this type of nonparametric analysis technique. [GIBBONS76]

CHAPTER 7 Addendum A

Cost Performance

We observed trends in both central tendency and in variation across the rating levels. The trend observed was high variation with central tendency below a CPI of 1.00 for Level 1, high variation and central tendency near a CPI of 1.00 for Level 2, and low variation and central tendency near a CPI of 1.00 for Level 3. Additionally, the multiple comparison test showed significant differences between Levels 1 and 2 and between Levels 1 and 3. Thus, the trend in CPI with increasing organization maturity is a CPI generally approaching 1.00 with generally decreasing variation (see Figure 7-16). The data show a pattern remarkably like that proposed by Paulk. [PAULK93]

Schedule Performance

For the schedule performance data, the variation appears fairly constant between Level 2 and Level 3 and is markedly less than the variation in SPI at Level 1. Thus, a trend in variation with rating level is shown only between Level 1 organizations and the rest. It appears that once an organization matures beyond Level 1, variation in SPI is relatively insensitive to maturity. Unlike the trend observed in variation, we observed no clear trend in the central tendency of SPI within the complete dataset.

At all rating levels, the SPI remains close to 1.00. However, when the *moderator "Percent Complete"* was taken into account, an intriguing correlation between rating level and central tendency of SPI manifested itself. We noted that for programs less than 80% complete, the performance of Level 1 organizations was consistently below an SPI of 1.00. Thus, the trend of increasing central tendency and decreasing variation with more mature organizations (similar to that observed in CPI) is observed only in programs less than 80% complete.

The trend is markedly different for programs greater than 80% complete, but this is probably attributable to the dynamics of the schedule performance index rather than a maturity effect. The SPI is driven — by definition — to a value of 1.00 at program completion. This results in predictably high SPIs for all programs near completion.

CONCLUSION

The aim of our research was to determine the nature of a correlation between the CMM[SM] rating and software development success. Though success is difficult to measure directly, by using the indicators of cost and schedule performance, we were able to show correlation between CMM[SM] rating and the cost and schedule performance of a generally representative sample of historical software development contracts.

We observed improved cost and schedule performance with increasing process maturity. Specifically, the least mature organizations were likely to have difficulty adhering to cost and schedule baselines. In contrast, the more mature organizations were likely to have on-baseline cost and schedule performance. This observed correlation was evident in the dataset as a whole, but was more evident in the sample which had high to very high rating-to-program relevance. The correlation was more evident in cost performance than in schedule performance.

This study has validated a correlation between program success and CMM[SM] ratings established in the same time frame as the program development. Although this result suggests that CMM[SM] ratings might be used as a predictor of future program success, more research is required before such a predictive relationship can be established. *[NOTE: See Volume 2, Appendix A for information on how to contact the authors: Patricia K. Lawlis, Captain Robert M Flowe (USAF), and Captain James B. Thordahl (USAF).]*

REFERENCES

[DEVORE82] Devore, J.L., Probability and Statistics for Engineering and Science, Third Edition, Brooks/Cole Publishing Company, Pacific Grove, California, 1982

[FLOWE94] Flowe, R.M., and J.B. Thordahl, "A Correlational Study of the SEI's Capability Maturity Model and Software Development Performance in DoD Contracts," master's thesis, Air Force Institute of Technology, December 1994

[GIBBONS76] Gibbons, J.D., Nonparametric Methods for Quantitative Analysis, Holt, Rinehart, and Winston, Chicago, 1976

[HUMPHREY90] Humphrey, W.S., Managing the Software Process, Addison-Wesley, Reading, Massachusetts, 1990

[MOSEMANN94] Mosemann, Lloyd K., II, "Why the New Metrics Policy?" *CrossTalk*, April 1994

[NICHOLAS90] Nicholas, J.M, Managing Business and Engineering Projects, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[PAULK93] Paulk, M.C., B. Curtis, M.B. Chrissis, and C.V. Weber, "Capability Maturity Model, Version 1.1," *IEEE Software*, Vol. 10, No. 7, July 1993

About the Authors

Patricia K. Lawlis is president and senior software engineering consultant for cj. kemp systems, inc. She holds a doctorate in computer science from Arizona State University and is a retired Air Force lieutenant colonel. Dr. Lawlis has worked in many areas of software engineering, including software development in many different computer languages. For 10 years she was on the regular faculty of the Air Force Institute of Technology where she continues to be an adjunct assistant professor of software engineering. She was one of the faculty advisers for the thesis work reported in this article. Dr. Lawlis has also been involved with extensive research in developing an object-oriented software architecture for the visual simulation application domain, and she is known for her work in language comparisons.

Patricia K. Lawlis, Ph.D.
c.j. kemp systems, inc.
P.O. Box 24363
Huber Heights, OH 45424
Phone: (513) 878-3303
Fax: (513) 878-3303
E-mail: lawlis@afit.af.mil; lawlis@aol.com

Capt. Robert M. Flowe has been a member of the Air Force since 1983, where he has served in operational support of the Titan II, III, and IV space launch vehicles. He received a master's degree in software systems management from the Air Force Institute of Technology, and this article is a result of his joint thesis work with Capt. James Thordahl in that degree program.

Capt. Robert M Flowe
Electronic Systems Center Dept. 44
Kelly AFB, TX 78243
Phone: (210) 977-3445; DSN 969-3445

CHAPTER 7 Addendum A

Capt. James B. Thordahl has been a member of the Air Force since 1988, where he has served in support of high-energy laser development for the Ground-Based Laser Antisatellite Program as well as in support of optical research. He received a master's degree in software systems management from the Air Force Institute of Technology.

Capt. James B. Thordahl
HQ SMC/MC3
2420 Vela Way, Suite 1467-A8
Los Angeles AFB, CA 90245-4659
Phone: (310) 336-2070; DSN 833-2070
Fax: (310) 336-4848; DSN 833-4848
E-mail: thordahl@mc.laafb.af.mil

CHAPTER 7
Addendum B

Lessons-Learned While Achieving a CMM[SM] Level 3 Rating

Sacramento Air Logistics Center

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

Blank page.

CHAPTER

8

Measurement and Metrics

CHAPTER OVERVIEW

Why is it so important to measure your process and the product it produces? Unlike other manufacturing processes, software development is inherently unstable. Any human-intensive activity, without control, deteriorates with time. It takes constant attention and discipline to keep the software production process from breaking down — let alone to improve it. If you do not measure, there is no way to know whether the process is on track or if it is improving. ***Measurement*** *provides a way for you to assess the status of your program to determine if it is in trouble, in need of corrective action, and/or process improvement. This assessment must be based on up-to-date measures that reflect current program status, both in relation to the program plan and to models of expected performance drawn from historical data of similar programs. If, through measurement, you diagnose your program as being in trouble, you will be able take meaningful, effective remedial action (e.g., relaxing performance requirements, extending your schedule, adding more money, or any number of options). [See Chapter 16, The Challenge, for a discussion on remedial actions for troubled programs.] Measurement provides benefits at the strategic, program, and technical levels.*

Software quality measures are often no more than counts of defects in the code. By measuring defects, you gain information for making decisions about rework, whether to release to the next unit or phase, to move intermediate code forward to the next development activity, or to continue the current activity. Defects must be identified, tracked, and resolved through software problem reports (SPRs) which are subject to rigorous configuration management rules. The ***defect discovery*** *and* ***resolution rate*** *is a direct measure of software process health.*

Poor size estimation is one of the main reasons major software-intensive acquisition programs ultimately fail. Size is the critical factor in determining cost, schedule, and effort, the failure to accurately predict (usually too small), results in budget overruns and late deliveries which undermine confidence and erode support for your program. Size estimation is a complicated activity the results of which must be constantly updated throughout the life cycle with actual counts. Size measures include source lines-of-code, function points, and feature points. Complexity is a function

of size which greatly impacts on design errors and latent defects which ultimately result in quality problems, cost overruns, and schedule slips. Complexity continuously must be measured, tracked, and controlled. Another factor leading to size growth is requirements creep which also must be baselined and diligently controlled.

Actual versus planned staff-hours expended should be tracked from the day of contract award. Effort expended is often the largest (and least controllable) cost variable. ***Effort measures*** *permit the comparison between planned (piece-of-cake) and actual (this-is-a-bit-more-difficult-than-we-thought) level of manhour expenditures which give us insights into productivity, another key cost driver. You will want to breakdown the various labor and support staff expenditures into task areas, such as levels of experience and task assignment. Again,* ***metrics reveal trends.*** *If the effort expended exceeds planned estimates, and cost or schedule begin to slide, watch for quality to follow.*

Scrap and rework measures assess the amount of effort lost when portions of the product thrown out or recoded because it does not meet expectations. Usually, scrap and rework occurs because defects are found in the code, or the product does not perform as required. ***Scrap and rework is a major variable in both cost and schedule.*** *Perhaps more than any other metric, scrap and rework reveals the contractor's software development maturity. If proper front-end planning, design, and code production are accomplished with quality as the underlying goal, scrap and rework should be a very small percent of total program effort.*

A good measurement program is an investment in success by facilitating early detection of problems, and by providing quantitative clarification of critical development issues. Metrics give you the ability to identify, resolve, and/or curtail risk issues before they surface. Measurement must not be a goal in itself. It must be integrated into the total software life cycle — not independent of it. To be effective, metrics must not only be collected — ***they must be used!*** *Campbell and Koster summed up the bottom line for metrics when they exclaimed:*

> *If you ain't measurin,' you ain't managin' — you're only along for the ride (downhill)!* [CAMPBELL95]

CHAPTER 8

Measurement and Metrics

MEASUREMENT

> *Measurement is the key to progress in software...Now that accurate measurements and metrics are available, it can be asserted that software engineering is ready to take its place beside the older engineering disciplines as a true profession, rather than an art or craft as it has been for so long.* [JONES91]

You cannot build quality software, or improve your process, without **measurement**. Measurement aids in achieving the basic management objectives of prediction, progress, and process improvement. An oft repeated phrase by **DeMarco** holds true, "*You can't manage what you can't measure!*" [DeMARCO86] All process improvement must be based on measuring where you have been, where you are now, and predicting where you are heading. ***Good metrics always lead to process improvement!***

Measures, Metrics, and Indicators

A software **measurement** is a quantifiable dimension, attribute, or amount of any aspect of a software program, product, or process. It is the raw data which identify various elements of the software process and product. **Metrics** are computed from measures. They are quantifiable indices used to compare software products, processes, or projects or to predict their outcomes. With metrics, we can:

- **Monitor** requirements,
- **Predict** development resources,

- **Track** development progress, and
- **Understand** maintenance costs.

A distinction should be made between **indicators** and **metrics**. With indicators, the requirement for determining a relationship between the value of the indicator and the value of the software characteristic being measured is substantially relaxed. A **reliability indicator**, for example, will not describe an anticipated value of reliability [i.e., mean-time-between-failure (MTBF) of 0.8 on a scale of 0 to 1]. Indicators are used to compare the current state of your program with past performance or prior estimates and are derived from earlier data from within the program. They show *trends* of increasing or decreasing values, relative only to the previous value of the same indicator. They also show containment or breeches of pre-established limits, such as allowable latent defects. Useful insights can be drawn from indicators because they are derived from readily available data internal to the program and do not require significant investment in resources or imposition on existing processes. Table 8-1 gives some examples of useful management indicators.

AREA	INDICATORS
Requirements	CSCI requirements
	CSCI design stability
Performance	Input/output bus throughput capability
	Processor memory utilization
	Processor throughput utilization
Schedule	Requirements allocation status
	Preliminary design status
	Code and unit test status
	Integration status
Cost	Person-months of effort
	Software size

Table 8-1 Example Management Indicators

A **metric**, on the other hand, is a direct measure of a software product that is embedded in a hierarchy of relationships ultimately connecting that metric with the software characteristic being measured. The

change in the value of the metric has a direct relationship to any change in the value of the software attribute (characteristic) being measured. Because they compare your program attributes with industry benchmarks and norms, they require access to historical databases of normalized information to provide insights into deviations from accepted levels of performance. Metrics provide key insights into how your program is performing, whether it is stable, and about the quality of your process.

Metrics are also useful for determining a *"business strategy"* (how resources are being used and consumed). For example, in producing hardware, management looks at a set of metrics for **scrap** and **rework**. From a software standpoint, you will want to see the same information on how much money, time, and manpower the process consumes that does not contribute to the end product. One way a software program might consume too many resources is if errors made in the requirements phase were not discovered and corrected until the coding phase. Not only does this create rework, but the ***cost to correct an error during the coding phase that was inserted during requirements definition is approximately 50% more costly to correct than one inserted and corrected during the coding phase***. [BOEHM81] The key is to catch errors as soon as possible (i.e., in the same phase that they are induced). *[See Chapter 15, Managing for Process Improvement, for a detailed discussion on error prevention.]*

Management metrics are measurements that help evaluate how well the contractor is proceeding in accomplishing their **Software Development Plan**. Trends in management metrics support forecasts of future progress, early trouble detection, and realism in plan adjustments. Software **product attributes** are measured to arrive at **product metrics** which determine user satisfaction with the delivered product or service. From the user's perspective, product attributes can be reliability, ease-of-use, timeliness, technical support, responsiveness, problem domain knowledge and understanding, and effectiveness (creative solution to the problem domain). Product attributes are measured to evaluate software quality factors, such as efficiency, integrity, reliability, survivability, usability, correctness, maintainability, verifiability, expandability, flexibility, portability, reusability, or interoperability. **Process metrics** are used to gauge organizations, tools, techniques, and procedures used to develop and deliver software products. [PRESSMAN92] **Process attributes** are measured to determine the status of each phase of development (from

requirements analysis to user acceptance) and of resources (dollars, people, and schedule) that impact each phase.

NOTE: Despite the SEI's Software Capability Evaluation (SCE) methods, process efficiency can vary widely within companies rated at the same maturity levels and from program to program.

There are five classes of metrics from generally used from a commercial perspective to measure the quantity and quality of software. During development technical and defect metrics are used. After market metrics are then collected which include user satisfaction, warranty, and reputation.

- **Technical metrics** are used to determine whether the code is well-structured, that manual for hardware and software use are adequate, that documentation is complete, correct, and up-to-date. Technical metrics also describe the external characteristics of the system's implementation.
- **Defect metrics** are used to determine that the system does not erroneously process data, does not abnormally terminate, and does not do the many other things associated with the failure of a software-intensive system.
- **End-user satisfaction** metrics are used to describe the (demand) value received from using the system.
- **Warranty metrics** reflect specific revenues and expenditures associated with correcting software defects on a case-by-case basis. These metrics are influenced by the level of defects, willingness of users to come forth with complaints, and the willingness and ability of the software developer to accommodate the user.
- **Reputation metrics** are used to assess perceived user satisfaction with the software and may generate the most value, since it can strongly influence what software is acquired. Reputation may differ significantly from actual satisfaction:
  - Because individual users may use only a small fraction of the functions provided in any software package; and
  - Because marketing and advertising often influences buyer perceptions of software quality more than actual use.

Software Measurement Life Cycle

Effective software measurement adds value to all life cycle phases. Figure 8-1 illustrates the primary measures associated with each phase of development. During the requirements phase, function

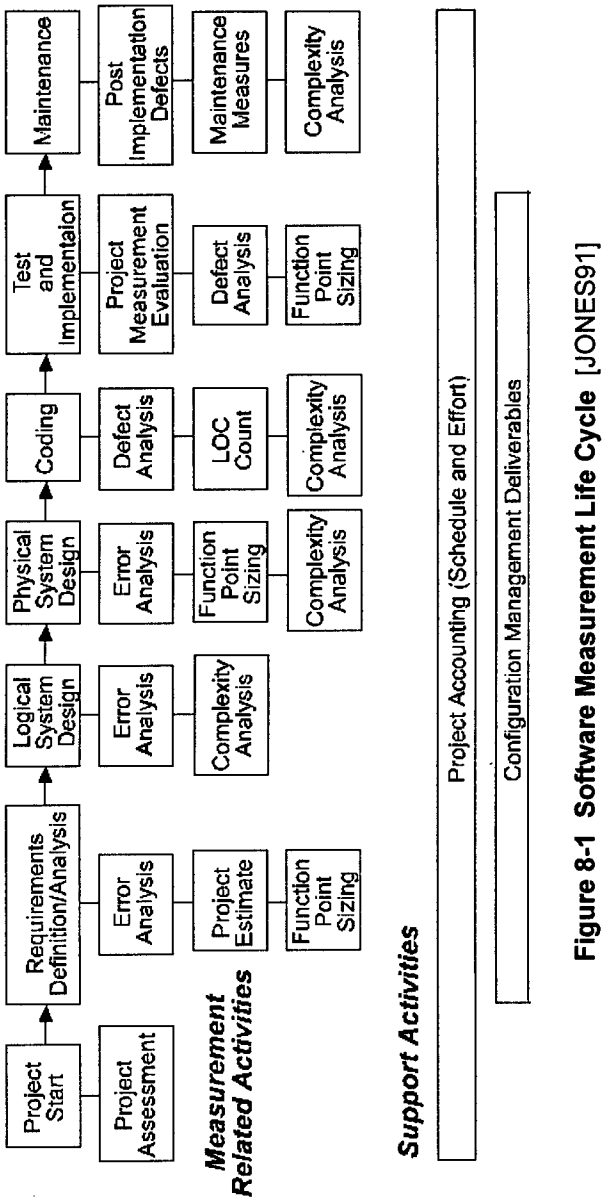


Figure 8-1 Software Measurement Life Cycle [JONES91]

points are counted. Since requirements problems are a major source of cost and schedule overruns, an error and quality tracking system is put into effect. Once requirements are defined, the next phase of measurement depends on whether the program is a custom development, or a combination of newly developed applications, reusable assets, and COTS. The risks and values of all candidate approaches are quantified and analyzed.

If the solution is a custom development, from the logical system design on, defect removal is the most costly activity. Therefore, design reviews and peer inspections *[discussed in Chapter 15, Managing Process Improvement]* are used to gather and record error data. During physical design, reviews and peer inspections continue to be valuable and error data are still collected. The coding phase can either be bothersome, or almost effortless, depending upon the success of the preceding phases. Defect and quality data, as well as code complexity measures, are recorded during code reviews and peer inspections.

One of the most important characteristics of your software development is ***complexity***. With the first executable architecture, throughout the design phases, and subsequently, as the code is developed and compiled, evaluate the complexity of your proposed development. This will give you important insight into the feasibility of bringing your program to a successful conclusion (i.e., to provide the desired functional performance on the predicted schedule, at the estimated cost).

The testing phase can range from a simple unit test a programmer conducts, to a full-scale, multi-staged formal test suite including function tests, integration tests, stress tests, regression tests, independent tests, field tests, system tests, and final acceptance tests. During all these activities, in-depth defect data are collected and analyzed for use in subsequent **defect prevention** exercises. Retrospective analyses are performed on **defect removal efficiencies** *[also discussed in Chapter 15, Managing Process Improvement]* for each specific review, peer inspection, and test, and on the cumulative efficiency of the overall series of defect removal steps.

During the maintenance phase, both user satisfaction and latent defect data are collected. For enhancements or replacements of existing systems, the structure, complexity, and defect rates of the existing software are determined. Further retrospective analysis of defect removal efficiency is also be performed. ***A general defect removal***

goal is a cumulative defect removal efficiency of 95% for MIS, and 99.9% (or greater) for real-time weapon systems. This means that, when defects are found by the development team or by the users, they are summed after the first (and subsequent) year(s) of operations. Ideally, the development team will find and remove 95% to 100% of all latent defects. [JONES91]

Software Measurement Life Cycle at Loral

Table 8-2 illustrates how **Loral Federal Systems**, Manassas collects and uses metrics throughout the software life cycle. The measurements collected and the metrics (or in-process indicators) derived from the measures change throughout development. At the start of a program (during the proposal phase or shortly after contract start), detailed development plans are established. These plans provide planned start and completion dates for each CDRL (CSU or CSCI). From these plans, a profile is developed showing the planned percent completion per month over the life of the program for all the software to be developed. Also at program start, a **launch meeting** is conducted to orient the team to a common development process and to present lessons-learned from previous programs.

PROGRAM PHASE	MEASUREMENT COLLECTION	IN-PROCESS INDICATORS (METRICS)
At Start	Establish SDP (plan start/completion for each phase); Establish initial measurements (e.g., attributes, code, labor)	Initial plan-complete profiles
During Design, Coding, Unit Testing	Weekly: Update and complete plan items Regularly: Inspection defect data Bi-monthly: Causal analysis meetings Quarterly: Update measurements	% plan versus % actual; Code change versus time Inspection effectiveness (defects found/ defects remaining); Defects/ KSLOC Process improvements identified Product quality (projected); Staffing profile
During Integration Testing	Program Trouble Reports (PTRs) Development team survey transferred to integration and test team survey	PTR density (PTRs/ KSLOC); Open PTRs versus time; % PTRs satisfied and survey comments
At End	Document lessons-learned Customer survey PTRs during acceptance	 % satisfied and customer comments used for future improvement Product quality (actual)

Table 8-2 Collection and Use of Metrics at Loral

During software design, coding, and unit testing, many measurements and metrics are used. On a weekly basis, actual percent completion status is collected against the plan established at program start. Actuals are then plotted against the plan to obtain early visibility into any variances. The number of weeks (early or late) for each line item is also tracked to determine where problems exist. Source statement counts (starting as estimates at program start) are updated whenever peer inspections are performed. A plot of code change over time is then produced. Peer inspections are conducted on a regular basis and defects are collected. **Peer inspection metrics** include **peer inspection efficiency** (effectiveness) (percent of defects detected during inspections versus those found later) and expected **product quality** [the number of defects detected per thousand source lines-of-code (KSLOC)]. Inspection data are used to project the expected latent defect rate (or conversely, the **defect removal efficiency rate**) after delivery. At the completion of each development phase, **defect causal analysis** meetings are held to examine detected defects and to establish procedures, which when implemented, will prevent similar defects from occurring in the future.

During development, program measurements are updated periodically (either monthly or quarterly) in the site metrics database. These measurements include data such as cost, effort, quality, risks, and technical performance measures. Other metrics, such as planned versus actual staffing profiles, can be derived from these data. During integration and test, trouble report data are collected. Two key metrics are produced during this phase: the **Program Trouble Report (PTR)** density (the number of defects per KSLOC) and **Open PTRs** over time. An internal development team survey is transferred to the integration and test team so they can derive the team's defect removal efficiency and PTR satisfaction for the internal software delivery.

At the end of the program, lessons-learned for process improvement are documented for use in future team launches. If possible, a customer survey is conducted to determine a **customer satisfaction rating** for the program. Delivered product quality is measured during acceptance testing and compared to the earlier projected quality rate. These actuals are then used to calibrate the projection model for future and on-going programs.

SOFTWARE MEASUREMENT PROCESS

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind...* — Lord Kelvin

Your **software measurement process** must be an objective, orderly method for quantifying, assessing, adjusting, and ultimately improving your development process. Data are collected based on known, or anticipated, development issues, concerns, and questions. They are then analyzed with respect to the software development process and products. The measurement process is used to assess quality, progress, and performance throughout all life cycle phases. The key components of an effective measurement process are:

- Clearly defined **software development issues** and the measure (data elements) needed to provide insight into those issues;
- **Processing of collected data** into graphical or tabular reports (indicators) to aid in issue analysis;
- **Analysis of indicators** to provide insight into development issues; and,
- Use of analysis results to **implement process improvements** and identify new issues and problems.

Your normal avenue for obtaining measurement data is via contract CDRLs. A prudent, Level 3 contractor will implement a measurement process even without government direction. This measurement process includes collecting and receiving actual data (not just graphs or indicators), and analyzing those data. To some extent the government program office can also implement a measurement process independent of the contractor's, especially if the contractor is not sufficiently mature to collect and analyze data on his own. In any case, it is important for the Government and the contractor to meet and discuss analysis results. ***Measurement activities keep you actively involved in, and in control of, all phases of the development process.*** Figure 8-2 (below) illustrates how measurement is integrated into the organizational hierarchy at IBM-Houston. It shows how a bottoms-up measurement process is folded into corporate activities for achieving corporate objectives.

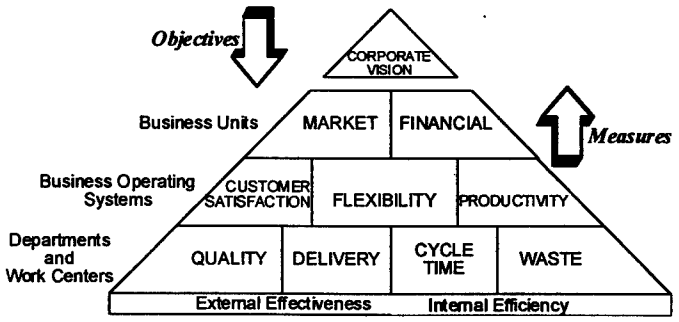


Figure 8-2 Organizational Measurement Hierarchy

Metrics Usage Plan

For measurement to be effective, it must become an integral part of your decision-making process. Insights gained from metrics should be merged with process knowledge gathered from other sources in the conduct of daily program activities. It is the ***entire measurement process*** that gives value-added to decision-making, not just the charts and reports. [ROZUM92] Without a firm **Metrics Usage Plan**, based on issue analysis, you can become overwhelmed by statistics, charts, graphs, and briefings to the point where you have little time for anything other than ingestion. ***Plan well!*** Not all data is worth collecting and analyzing. Once your development program is *in-process,* and your development team begins to design and produce lines-of-code, the effort involved in planning and specifying the metrics to be collected, analyzed, and reported upon begins to pay dividends. Figure 8-3 illustrates examples of life cycle measures and their benefits collected on the **Space Shuttle** program.

The ground rules for a Metrics Usage Plan are that:

- **Metrics must be understandable to be useful**. For example, lines-of-code and function points are the most common, accepted measures of software size with which software engineers are most familiar.
- **Metrics must be economical**. Metrics must be available as a natural by-product of the work itself and integral to the software development process. Studies indicate that approximately 5% to 10% of total software development costs can be spent on metrics. The larger the software program, the more valuable the investment in metrics becomes. Therefore, do not waste programmer time by requiring specialty data collection that interferes with the coding

CHAPTER 8 Measurement and Metrics

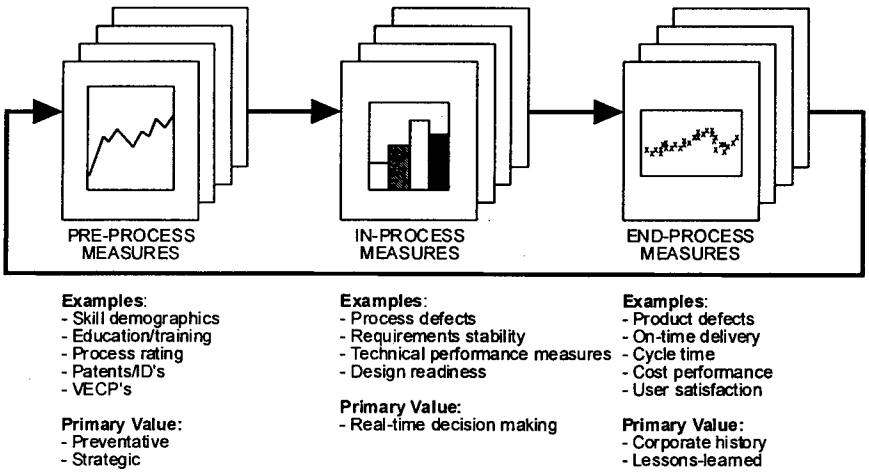


Figure 8-3 Space Shuttle Life Cycle Measurements

task. Look for tools *[discussed in Chapter 10, Software Tools]* which can collect most data on an unintrusive basis.

- **Metrics must be field tested**. Beware of software contractors who offer metrics programs that appear to have a sound theoretical basis, but have not had practical application or evaluation.
- **Metrics must be highly leveraged**. You are looking for data about the software development process that permit management to make significant improvements. Metrics that show deviations of .005% should be relegated to the trivia bin.
- **Metrics must be timely**. If a measurement is not available until the program is in deep trouble because, for instance, defect detection and repair rates were not measured early on, *you have landed long and the overrun is upon you.* [YOURDON92]
- **Metrics must give proper incentives for process improvement**. High scoring teams are driven to improve performance when trends of increasing improvement and past successes are quantified. Conversely, metrics data should be used very carefully during contractor performance reviews. A poor performance review, based on metrics data, can lead to negative government/industry working relationships. ***Remember, do not use metrics to judge team or individual performance.***
- **Metrics must be evenly spaced** throughout all phases of development. Effective measurement adds value to all life cycle activities. [JONES91]
- **Metrics must be useful at multiple levels**. They must be meaningful to both management and technical team members for process improvement of all facets of development.

Boeing 777 Metrics Program

According to **Ronald J. Pehrson**, manager of Boeing's Embedded Software Engineering Commerical Airplane Group, the software metrics used on the **Boeing 777** development program tracked progress against plans for design, code, test procedure creation, and test completion. They included the predicted total software size (in SLOC) and total number of tests. Metrics charts showed key milestones in the airplane program. These milestones represented interim points to measure progress against the ultimate program completion. Associated with these milestones, were success criteria based on completion of software product design, code, and testing. Figure 8-4 shows the total airplane metrics program roll-up. *[Their metrics also measured utilization of computer resources (throughput and memory) though not discussed here.]* Boeing's metrics process included the following:

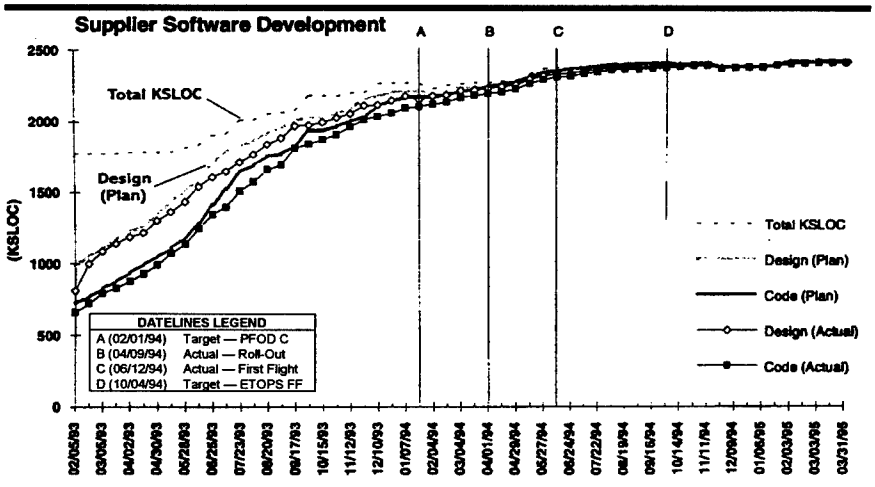


Figure 8-4 Total Metrics Roll-Up for the 777 [PEHRSON96]

- **Supplier Metrics Plans**. Each supplier was requested to prepare plans for their design, code, and test activities. These plans showed expected totals and the planned completion status for each biweekly reporting period until the task was complete. Even at this early stage in the metrics process, they achieved their first benefits as they discovered that some suppliers' initial plans did not support program milestones. This proved invaluable insight, and in a few cases was the only major corrective action needed to assure the supplier supported the program.

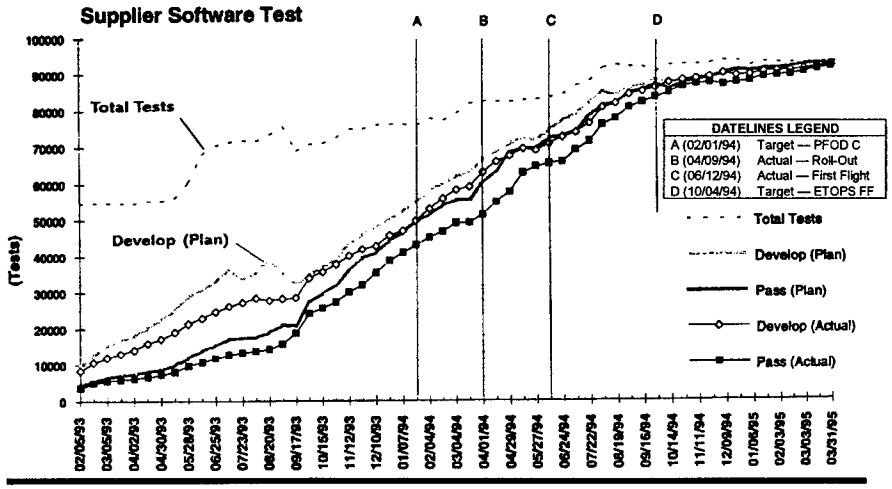


Figure 8-4 Total Metrics Roll-Up for the 777 (cont.) [PEHRSON96]

- **Biweekly plan updates**. Following initial plan submittal, biweekly updates showed actual development status in terms of completed design, code, and tests. Any changes in the estimated total size of the effort were also provided and the plans modified to correctly reflect new totals. Plans could be changed at any time; however, previously reported plans and actual status were not adjusted. The metrics information was shared with the systems developers. This led to important discussions on how they were going to succeed at an early enough point in the program where they could actually do something about it.
- **Metrics used for process improvement**. Indications of a healthy program were fairly obvious — a plan that supported program milestones and status that consistently tracked to the plan. Programs needing special attention were often several weeks behind the plan line, had numerous re-plans, or had plans that required unprecedented productivity to be successful. They quickly established reasonable productivity figures that easily tested the feasibility of suppliers' plans based simply on head count, work to go, and time to go. The metrics provided an excellent vehicle for discussions about how the program was going. They helped in dealing with their current status and in getting back on a schedule that supported the overall airplane program.

The above metrics process was as important as the measures themselves in assuring success. However, there were certain characteristics of the metrics program that were key to supporting this process and making it all work. They were as follows:

- Uniformity,
- Frequent updates,
- Clear definition,
- Objective measures,
- Re-plans, as needed, were allowed and even encouraged, and
- Past plans and actuals were held constant.

The uniform nature of the metrics program enabled comparison across systems and supported communication of objective status to all levels of program management. This was particularly important with the large number of organizations involved in the software development. Boeing was also able to combine status information from several different systems provided by a single supplier. This provided unique opportunities to discuss how the supplier was supporting the overall program and to focus needed resources to solve schedule problems.

Considerable effort was given to clearly define measures. This led to a 21-page set of instructions to their suppliers on how to prepare metrics data. The data items measured were objective and easily observed. The combination of these meant there was little confusion about what the metrics meant so real conclusions could be drawn from the data. Moreover, without this they would not have achieved the desired uniformity. Two aspects of the metrics plans were critical: replanning when needed was encouraged, and past data were never changed. The essence of a plan is it shows how to get from here to there. Once you have significantly deviated from a plan it no longer serves that purpose. Throughout the metrics process, Boeing used *deviation from the plan* as an indicator of problems. Since replanning was encouraged, the only reason to not be close to their plan was they did not have a plan. They found that programs several weeks behind their plan did indeed need help.

This approach to software program metrics repeatedly "*saved their bacon.*" Starting with initial plans, they indicated where program milestones were not being supported. Continuous monitoring through testing identified schedule problems early in the development process. The metrics were invaluable in showing where program risk points were soon enough to take corrective action. [PEHRSON96]

Metrics Selection

Marciniak and Reifer proclaim that: "*Software projects don't get into trouble all at once; instead they get into trouble a little at a time.*" [MARCINIAK90] As illustrated on the Boeing 777 program, metrics must be selected to ascertain your program's ***trouble threshold*** at the earliest phase of development. Basic measures should be tailored by defining and collecting data that address those trouble (risk) areas identified in the **Risk Management Plan**. ***A rule of thumb for metrics is that they must provide insight into areas needing process improvement!*** Which metrics to use depends on the maturity level of the organization. Table 8-3 illustrates how metrics are used to obtain program knowledge, based on the Army's STEP metrics process. *[See Army policy on Preparation for Implementing Army Software Test and Evaluation Panel (STEP) Metrics Recommendations, Volume 2, Appendix C.]*

METRIC	OBJECTIVE
Schedule	Track progress versus schedule
Cost	Track software expenditures
Computer resource utilization	Track planned versus actual size
Software engineering environment	Rate contractor environment
Design stability	Rate stability of software design and planned release or block. Design stability = [(total modules+deleted module+modules with design changes)/total modules]. Design tresholds: >=0.9 (satisfactory), <0.9 to <0.85 (warning), <=0.85 (alarm)
Requirements traceability	Track rqmts to code by showing % of software reqmts traceable to developer specifications, config items (CI), CSCI, or CSC. Traceability tresholds are: >=99.9% (satisfactory), <99.9% to <99.5% (warning), <=99.5% (alarm)
Requirements stability	Track changes to requirements
Fault profiles	Track open versus closed anomalies
Complexity	Assess code quality
Breadth of testing	Extent to which reqmts are tested. 4 measures are: coverage (ratio of rqmts tested to total rqmts); test success (ratio of rqmts passed to rqmts tested); overall success (ratio rqmts passed to total rqmts); deferred (total deferred rqmts). Overall success thresholds: >=99% (satisfactory), <99% to <95% (warning), <=95% (alarm)
Depth of testing	Track testing of code
Reliability	Monitor potential downtime due to software

Table 8-3 How Metrics Are Used for Program Management [ARMY]

ATTENTION! Metrics selection must focus on those areas you have identified as sources of significant risk for your program.

NOTE: Examples of minimum measures include: quarterly collation and analysis of size (counting source statements and/or function/feature points), effort (counting staff hours by task and nature of work; e.g., hours for peer inspection), schedule (software events over time), software quality (problems, failures, and faults), rework (SLOC changed or abandoned), requirements traceability (percent of requirements traced to design, code, and test), complexity (quality of code), and breadth of testing (degree of code testing).

[Contact the Global Transportation Network (GTN) Program for a copy of the GTN Software Development Metrics Guidelines which discusses the metrics selected and gives examples of how to understand and interpret them. See Volume 2, Appendix A for information on how to contact the GTN program office.]

Data Collection

Get your facts first... then you can distort 'em as much as you please. — Mark Twain

As stated above, metrics are representations of the software and the software development process that produces them — the more mature the software development process, the more advanced the metrics process. A well-managed process, with a well-defined data collection effort embedded within it, provides better data and more reliable metrics. ***Accurate data collection is the basis of a good metrics process.*** You must, therefore, determine what data to collect, and how to define those data based both on current and projected program issues and on the characteristics of your software development process and products. Figure 8-5 illustrates the variety of software quality metrics and management indicators that were collected and tracked for all ATF (F-22) weapon systems functions.

After you have identified your program issues (and before contract award) you and your future contractor must agree on **entry** and **exit criteria** definitions for the proposed software development process and products. Entry and exit criteria must also be defined for all data

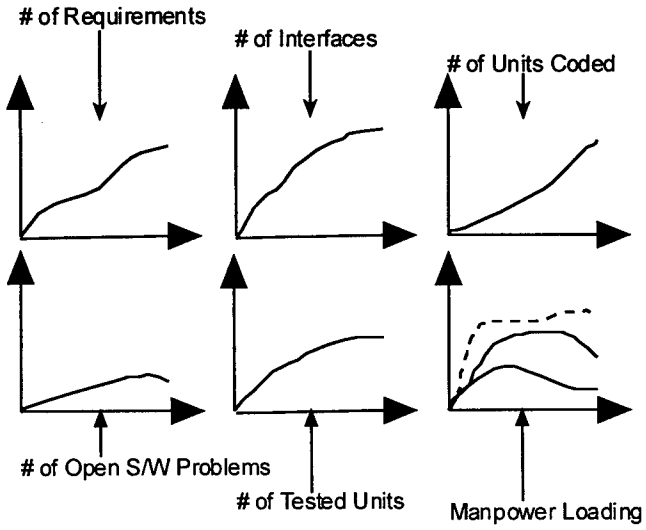


Figure 8-5 ATF Data Collection for Software Development Tracking

inputs, standards of acceptance, schedule and progress estimation, and data collection and analysis methods. For instance, there must be an agreement on the definition of source lines-of-code and how and when SLOC will be estimated or counted. ***The entire collection and analysis process — all definitions, decisions, and agreements — should be written into the contract.*** Once data collection has commenced, it is important that there is consistency in data definitions. Changing a definition midstream during data collection produces variations in data trends that can skew the analysis of performance, quality, and related issues. If definitions do change through process-knowledge, it is critically important that you understand each change and how these changes will affect already collected data. *[Changes in entry/exit definitions should be reflected in an updated SDP.]*

To summarize, data collection must be woven into the developer's process. For instance, count the number of software units to be built by looking at the **Software Design Description (SDD)** and limit the use of *out-of-range* data. Also, avoid collecting data that are derived, ill-defined, or cannot be traced directly back to the process. As the development progresses, new tools or data collection techniques may emerge. If new data collection methods are employed, the data collection efforts must be tailored to these techniques. In addition, as the data collection changes, your understanding of what those data mean must also change. [ROZUM92]

Data Analysis

Both objective and subjective measures are important to consider when assessing the current state of your program. **Objective data** consists of actual item counts (e.g., staff hours, SLOC, function points, components, test items, units coded, changes, or errors) that can be independently verified. Objective data are collected through a formal data collection process. **Subjective data** are based on an individual's (or group's) feeling or understanding of a certain characteristic or condition (e.g., level of problem difficulty, degree of new technology involved, stability of requirements). Objective and subjective data together serve as a system of checks and balances throughout the life cycle. If you are a resourceful manager, you will depend on both to get an accurate picture of your program's health. Subjective data provide critical information for interpreting and validating objective data; while objective data provide true counts that may cause you to question your subjective understanding and investigate further.

Analysis of the collected data must determine which issues are being addressed, and if new issues have emerged. Before making decisions and taking action from the data, you must thoroughly understand what the metrics mean. To understand the data, you must:

- **Use multiple sources to validate the accuracy of your data** and to determine differences and causes in seemingly identical sets of data. For instance, when counting software defects by severity, spot check actual problem reports to make sure the definition of severity levels is being followed and properly recorded.
- **Study the lower-level data collection process**, understand what the data represent, and how they were measured.
- **Separate collected data and their related issues from program issues**. There will be issues about the data themselves (sometimes negating the use of certain data items). However, do not get bogged down in data issues. You should concentrate on program issues (and the data items in which you have confidence) to provide the desired insight.
- **Do not assume data from different sources (e.g., from SQA or subcontractors) are based on the same definitions,** even if predefined definitions have been established. You must re-verify definitions and identify any variations or differences in data from outside sources when using them for comparisons.

- **Realize development processes and products are dynamic and subject to change**. Periodic reassessment of your metrics program guarantees that it evolves. Metrics are only meaningful if they provide insight into your current list of prioritized issues and risks.

Metrics should be compared to get the whole picture of program status and progress. As illustrated in Figure 8-6, the metrics analyzed (chosen dependent on software development maturity level) can be interrelated and flow into each other.

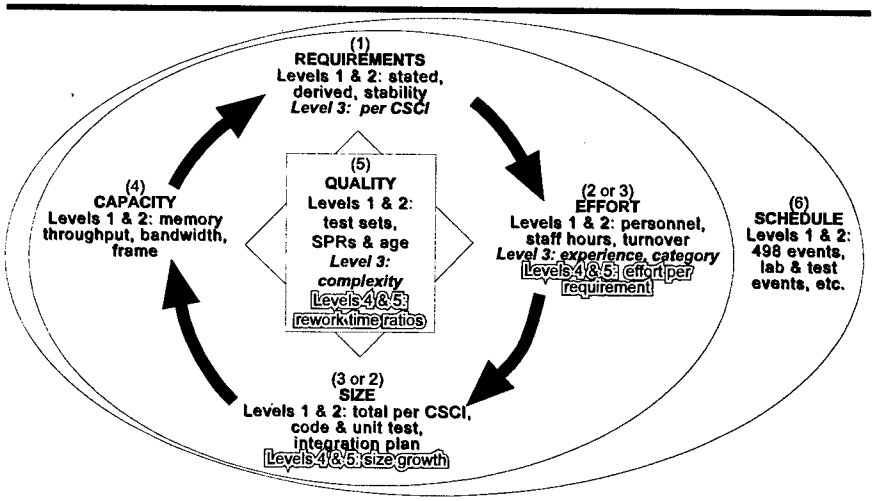


Figure 8-6 CMM Maturity Level, Measures Collected, Metrics Compared [CAMPBELL95]

The availability of program metrics is of little or no value unless you also have access to **models** (or norms) that represent what is expected. The metrics are used to collect historical data and experience. As their use and program input increases, they are used to generate databases of information that become increasingly more accurate and meaningful. Using information extracted from these databases, you are able to gauge whether measurement trends in your program differ from similar past programs and from expected models of optimum program performance within your software domain. Databases often contain key characteristics upon which models of performance are designed. **Cost data** usually reflect measures of effort. **Process data** usually reflect information about the programs (such as methodology, tools, and techniques used) and information about personnel experience

and training. **Product data** include size, change, and defect information and the results of statistical analyses of delivered code.

Figure 8-7 illustrates the possibilities for useful comparison by using metrics, based on available program histories. By using models based on completed software developments, the initiation and revision of your current plans and estimates will be based on ***"informed data."*** As you gather performance data on your program, you should compare your values with those for related programs in historical databases *[see NSDIR discussion below].* The comparisons in Figure 8-7 should be viewed collectively, as one component of a *feedback-and-control* system that leads to revisions in your management plan. To execute your revised plans, you must make improvements in your development process which will produce adjusted measures for the next round of comparisons.

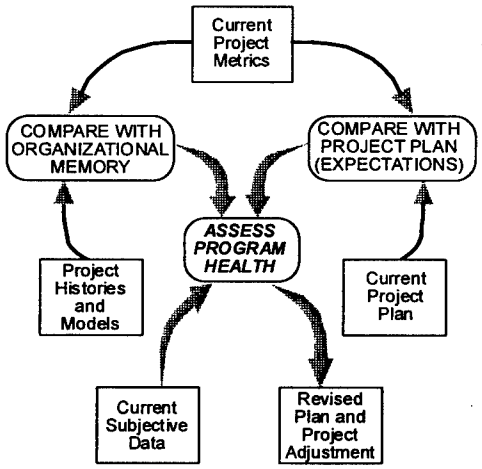


Figure 8-7 Management Process with Metrics

National Software Data and Information Repository (NSDIR)

The **National Software Data and Information Repository (NSDIR)** is a managed warehouse of data (and resulting information) about software development and maintenance programs. The repository collects, manages, and analyses the by-products of software engineering processes and products. Such by-products are measures about the:

- Program and product size,
- Program schedule,
- Nature of the program,
- Effort expended on the program,
- Quality of program products, and
- Other program attributes, such as rework, reuse, and practices.

The NSDIR, an Air Force sponsored program, is chartered to benchmark technologies and methodologies utilized in software-intensive programs through the collection and management of software product and process metrics data. Although initially focused on Air Force programs, the NSDIR is challenged to obtain like-metrics data from other sources. Data from the other services, federal agencies, and commercial industry enables decision-makers to answer questions such as, but not limited to:

- Degree of language use and methods, tools, techniques, and standards used,
- Relative measures of maintenance versus new development activity,
- Nature and type of software work contracted out,
- Size, scope, and nature of software engineering programs,
- Extent of open systems, COTS, GOTS, and reuse implementation, and
- Amount and type of documentation produced.

Overall, the NSDIR effort has two main objects: developing a repository capability and developing an information analysis capability. The basic **repository capabilities** are the collection, storage, and retrieval of metrics data. The repository evolves as the information needs of the users change. The enterprise-level database grew out of a need to focus on software measurement data which software development programs are likely to collect and report at natural intervals. The data requirements implemented by the **NSDIR Metrics Collection and Submission Guide** ensures a database contains information that is:

- Useful at an enterprise level for NSDIR users,
- Available from a large number of programs,
- Collected at significant program dates (milestones), and
- Suitable for benchmarking.

Data analysis capability focuses on data once entered into the repository. Users of the repository access these data from their desktop personal computers through various telecommunications methods. Through the use of standard desktop analysis and office automation tools, a user of the repository selects from a range of pre-defined queries. Varying these queries according to interest, the user performs analysis, reporting, and presentation of the data and information stored in the repository. Additionally, *ad hoc* queries are supported to enable the user to perform unanticipated analyses. With this analysis capability, repository users can generate graph-based reports for:

- Benchmarking their own program status and trends,
- Summary data across programs, and
- Trend data across programs.

Why the NSDIR Is Necessary

DoD faces rapidly changing budgetary, software technology, and military threat environments which it has responded to with initiatives to streamline acquisition, adopt commercial practices, and improve government and contractor processes. Guiding this transition requires accurate information about the nature of DoD software applications, the state of DoD software practice, and the performance of DoD software processes. DoD executives and program managers need quantitative answers to many difficult questions if to be successful in this challenging environment. Key questions about program planning asked by program managers include:

- How big is a typical program of this type,
- What is the typical productivity for this type of software,
- How many high-priority problems are found in a typical program,
- What level of rework effort can be expected for this type of software,
- How many people does it take, typically, to maintain 100,000 lines-of-code,
- How much code growth due to requirements change occurs in a typical program, and
- What are leading-edge programs doing in terms of methods and tools?

Getting answers to these questions helps managers and sponsors develop more realistic and achievable plans. Key policy and investment questions asked by DoD executives include:

- Are productivity and quality in my organization improving over time,
- What types of applications involve the most software,
- Which technologies offer the greatest potential for improving productivity and quality,
- How widely have technology innovations (e.g., Ada, process maturity assessments) been adopted,
- How predictable is the software process in terms of budget and schedule, and
- Where are other organizations investing their resources?

Getting answers to these questions helps the executive to define more effective policy, and to make better investments in training, research, and technology to support the organization. Currently, little data exist in the form of benchmarks and norms that can be used to:

- Define and promote "*best practices*,"
- Provide a reliable basis for estimates, and
- Assess the state of the software industry.

The NSDIR repository is needed to store and retrieve software data and information needed to effectively manage software engineering processes and enterprises. Because measurement is key to advancing to an "*engineering process*," the NSDIR is needed to supply the nation with professional information on "*best practices*" and software engineering benchmarks.

NSDIR History

There are two major challenges common to both commercial and defense software industries: (1) the transition of a current, state-of-the-art software engineering technology base into state-of-the-practice, and (2) the routine effective use of software measurement, data collection and analysis, and reporting practices. The transition from state-of-the-art software engineering technologies to state-of-the-practice must to be accelerated. The implementation of software measurement programs must become an integral and common component of the technical and organizational infrastructures of the US software industry to support a wide variety of management decisions, as well as process improvement. Successfully addressing these challenges will have a positive effect on the overall industry performance.

Statistically-based organizational process management, control, and improvement is an effective mechanism upon which an organization can base decisions changing and adopting enabling technologies to enhance quality and productivity. It is observed that overall the US software industry does a poor job of measuring software products, processes, and resources essential for improving the industry's overall quality and productivity.

Based on the current status of software measurement practices in the US and their importance to a highly-competitive and effective commercial and defense software industry, there is a critical need to establish a national capability for collecting and maintaining software measurement data. This national capability will provide the US software industry with an effective and low-risk approach for making decisions on inserting, modifying, and/or eliminating technologies employed for software development and maintenance. However, software measurement theory is very immature. Software metrics are primarily based on consensus rather than formal mathematical representation. This is not be surprising for a new discipline. Most disciplines begin applying subjective measures and evolve to objective ones upon maturing. Accelerating the software measurement maturation curve will provide industry with an effective foundation upon which to base technology improvement decisions. Thus, the NSDIR is needed to collect experience upon which to base objective software measurement and validation.

A national software repository must be part of an overall national vision. There must be a driving force to design and realize this vision and the vision must address software measurement issues at the national level. For example, the vision must address international competitiveness, cost-effectiveness, standards, certification, technology transfer, and technology gaps — to name a few.

In August 1993, over 60 leaders from industry, academia, and the Government participated in the first Software Measurement Workshop to discuss national-level software challenges. What is now referred to as Cooperstown I, resulted in an agreement to develop a strategy for creating the NSDIR and a blueprint for creating a National Software Council. A small operational prototype effort for the NSDIR began with initial funding from the US Air Force. The goal of the prototype, called Order-I, was to be a requirements elicitor and a mechanism for identifying technical and nontechnical barriers in establishing the NSDIR.

In August 1994, at Cooperstown II, eight working groups were formed to discuss and make recommendations concerning the evolution of the Order-I prototype towards the NSDIR, the establishment of a National Software Center, and requirements for an Information Base Repository. These recommendations for the NSDIR, combined with results from various technical interchange meetings of the NSDIR team, shaped the current focus of the program, termed NSDIR Phase II.

TYPICAL SOFTWARE MEASUREMENTS AND METRICS

A comprehensive list of industry metrics is available for software engineering management use, ranging from high-level effort and software size measures to detailed requirements measures and personnel information. ***Quality***, not quantity, should be the guiding factor in selecting metrics. It is best to choose a small, meaningful set of metrics that have solid baselines in a similar environment. A typical set of metrics might include:

- Quality,
  - User satisfaction,
- Size,
  - Source lines-of-code,
  - Function points,
  - Feature points,
- Complexity,
- Requirements,
- Effort,
- Productivity,
- Cost and schedule,
- Scrap and rework, and
- Support.

[Some industry sources of historical data are the NSDIR (discussed above) and those listed in Volume 2, Appendix A. A good source of information on optional software metrics and complexity measures are the SEI technical reports listed in Volume 2, Appendix E. Also see Rome Laboratory report, RL-TR-94-146, Framework Implementation Guidebook, for a discussion on software quality indicators. Another must-read reference is the Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report, USAMSAA, May 6, 1992 and policy memorandum, Preparation for Implementing Army

Software Test and Evaluation Panel (STEP) Metrics Recommendations, Volume 2, Appendix C.]

ATTENTION! ***Practical Software Measurement: A Guide to Objective Program Insight,*** **sponsored by the Joint Logistics Commanders, Joint Policy Coordinating Group on Computer Resources Management, is the recommended guide for setting up a measurement program for all major DoD software-intensive systems.** *[Information on how to obtain a copy is found in Volume 2, Appendices A and B.]*

NOTE: See Addendum A, *Assessment Metrics for Use with the Capability Maturity Model: Are We Improving?* See Volume 2, Appendix O, Chapter 8 Addendum C, "*Making Metrics Work Miracles*," to understand how setting up a metrics program can provide valuable program insights.

Quality

Measuring product quality is difficult for a number of reasons. One reason is the lack of a precise definition for quality. **Quality** can be defined as the ***degree of excellence*** that is measurable in your product. The IEEE definition for software quality is:

- The total features and characteristics of a software product that bear on its ability to satisfy given user needs; for example, conform to stated specifications;
- The degree to which the software possesses a desired combination of attributes;
- The degree to which a customer or user perceives that the software meets his or her composite expectations; and
- The composite characteristics of software that determine the degree to which the software, once in use, will meet the expectations of the user. [IEEE83]

Quality is in the eye of the user! For some programs, product quality might be defined as **reliability** [i.e., a low failure density (rate)], while on others **maintainability** is the requirement for a quality product. [MARCINIAK90] Your definition of a quality product must be based on **measurable quality attributes** that satisfy your program's specified user requirements. Because

requirements differ among programs, quality attributes will also vary. [MARCINIAK90]

Rome Laboratory's **1994 Framework Guidebook** lists a set of software quality attributes that are quantitatively measurable. These measures can be translated into terms understandable to the ultimate determinants of quality, the users and maintainers. Table 8-4 (below) lists the Rome's software quality factors, a definition, and a candidate metric for each. *[Do you see a correlation between the software engineering principles, discussed in Chapter 4, Engineering Software-Intensive Systems, and the attributes for quality software? Is this surprising?]*

User Satisfaction

John Gilligan, the Air Force Program Executive Official for Combat Support Systems, describes a set of user-focused metrics to measure system performance. Figure 8-8 (below) illustrates an example of mission availability on a C4I program. Data points reflect overall mission availability during quarterly exercises of the product by the user. The lower dark line (threshold) marks developer/user agreed upon unsatisfactory level of performance. The upper dark line (goal) marks developer/user agreed upon performance objective above the threshold, yet still achievable. The circles indicate system availability calculated from user (terminal) perspective (not CPU time) for accomplishing user mission tasks.

Figure 8-9 (below) illustrates user satisfaction in meeting warfighting supportability requirements for a nominal C4I system. The lower dark line (threshold) marks developer/user agreed upon unsatisfactory level of performance. The circles indicate actual, discrete data for mission support activity (e.g., spares, manpower, and depot support) per time period compared with the planned support.

Figure 8-10 (below) illustrates user satisfaction with a nominal C4I system's operating costs. The upper dashed line (threshold) marks developer/user agreed upon unsatisfactory level of operating costs per installation. The lower dashed line (goal) marks developer/user agreed upon operating costs below current level yet still achievable. Data points collected provide the applicable system's average total operating cost per installation. Those costs include: training, hardware/ software maintenance, license fees, communications, help desk, and central training costs. The circles indicate semiannual data collection points. [GILLIGAN94]

SOFTWARE QUALITY FACTOR	DEFINITION	CANDIDATE METRIC
Correctness	Extent to which the software conforms to specifications and standards	$Defects / LOC$
Efficiency	Relative extent to which a resource is utilized (i.e., storage, space, processing time, communication time)	$\frac{Actual\ ResourseUtilization}{Allocated\ ResourseUtilization}$
Expandability	Relative effort to increase software capability or performance by enhancing current functions or by adding new functions or data	$\frac{EffortToExpand}{EffortToDevelop}$
Flexibility	Ease of effort for changing software missions, functions, or data to satisfy other requirements	$(0.05)[AvgLaborDaysToChange]$
Integrity	Extent to which the software will perform without failure due to unauthorized access to the code or data	$Defects / LOC$
Interoperability	Relative effort to couple the software of one system to the software of another	$\frac{EffortToCouple}{EffortToDevelop}$
Maintainability	Ease of effort for locating and fixing a software failure within a specified time period	$(0.1)[AvgLaborDaysToFix]$
Portability	Relative effort to transport the software for use in another environment (hardware configuration, and/or software system environment)	$\frac{EffortToTransport}{EffortToDevelop}$
Reliability	Extent to which the software will perform without any failures within a specified time period	$Defects / LOC$
Reusability	Relative effort to convert a software component for use in another application	$\frac{EffortToConvert}{EffortToDevelop}$
Survivability	Extent to which the software will perform and support critical functions without failure within a specified time period when a portion of the system is inoperable	$Defects / LOC$
Usability	Relative effort for using software (training and operation, e.g., familiarization, input preparation, execution, output interpretation)	$\frac{LaborDaysToUse}{LaborYearsToDevelop}$
Verifiability	Relative effort to verify the specified software operation and performance	$\frac{EffortToVerify}{EffortToDevelop}$

Table 8-4 RADC Software Quality Factors

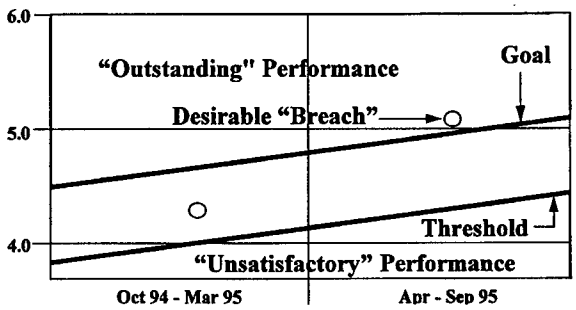


Figure 8-8 Mission Availability Satisfaction [GILLIGAN]

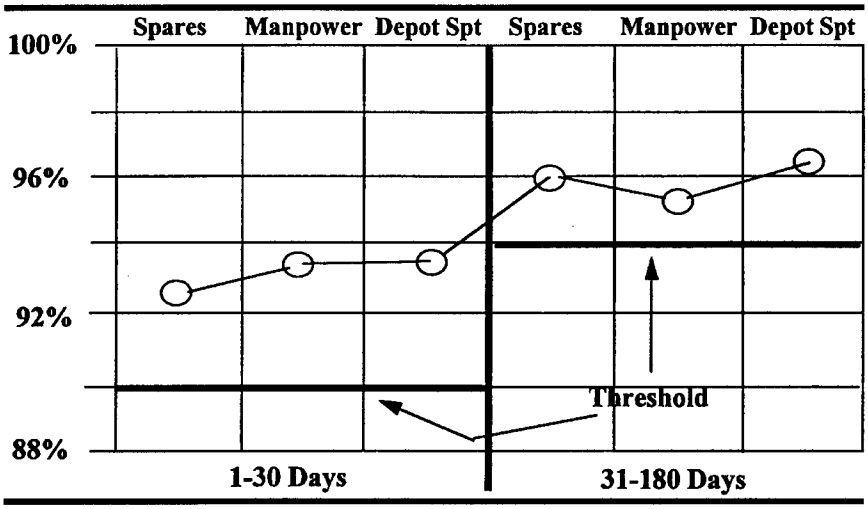


Figure 8-9 Warfighting Supportability Requirements Satisfaction [GILLIGAN94]

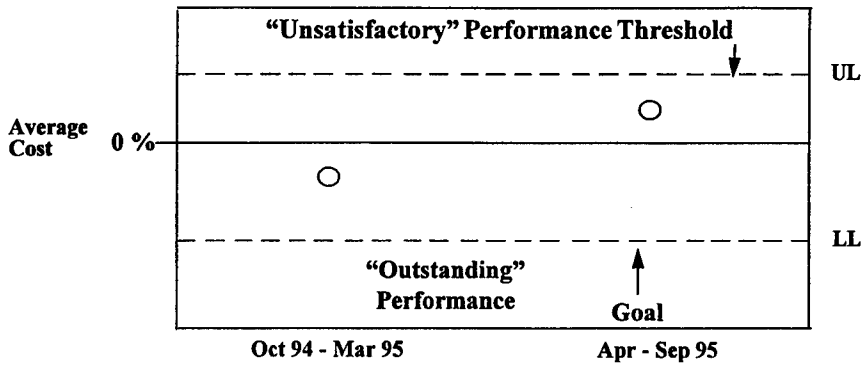


Figure 8-10 Operating Cost Satisfaction (total cost/installation) [GILLIGAN94]

Size

Just as we typically need to determine the weight, volume, and dynamic flight characteristics of a developmental aircraft as part of the planning process, you need to determine how much software to build. However, as you learned in Chapter 1, *Software Acquisition Overview*, one of the main reasons software programs fail is our inability to accurately estimate **software size**. Because we invariably estimate size too low, we do not adequately fund or allow enough time for development. Poor size estimates are usually at the heart of cost and schedule overruns. ***The trend has been to estimate size too small!***

There are two common types of size inaccuracies for which you can compensate to some degree. (1) Normal **statistical inaccuracies** can be dealt with by using multiple data sources and estimating methodologies, or by using multiple organizations to do the estimating and check and analyze results. (2) The earlier the estimate is made — the less is known about the software to be developed — the greater the estimating errors. Basing your estimates on more than one source is sound advice for both type of discrepancies. In addition, accuracy can be improved if estimates are performed at the smallest product element practical. Therefore, base your estimates on the smallest possible unit of each component. Then compile these calculations into composite figures. [HUMPHREY89] The key to credible software sizing is to use different software sizing techniques, and not to rely on a single source or method for the estimate. Reliance on a single source or technique represents a major contribution to program cost and schedule risk, especially if it originates exclusively from a contractor based on their bid.

Given our shortcomings in size estimation, it is absolutely critical that you measure, track, and control software size throughout development. You need to track the actual software size against original estimates (and revisions thereto) both incrementally and for the total build. Analysis is necessary to determine trends in software size and functionality progress. Data requirements for these measures are stated in contract CDRL items which include:

- The number of distinct functional requirements in the SRS and IRS,
- The number of software units contained in the SDP or SDD, and
- SLOC or function point estimates for each CSCI and build compared to the actual source code listing for each software unit.

CHAPTER 8 Measurement and Metrics

Software size has a direct effect on overall development cost and schedule. Early significant deviations in software size data indicate **problems** such as:

- Problems in the model(s), logic, and rationale used to develop the estimates,
- Problems in requirements stability, design, coding, and process,
- Unrealistic interpretation of original requirements and resource estimates to develop the system, and
- Faulty software productivity rate estimates.

Significant departures from code development estimates should trigger a risk assessment of the present and overall effort. Size-based models *[discussed in Chapter 10, Software Tools]* should be revisited to compare your development program with those of similar domain, scope, size, and complexity, if possible.

Measuring Software Size

There are two basic methods for measuring software size. Historically, the primary measure of software size has been the number **source lines-of-code (SLOC)**. However, it is difficult to relate software functional requirements to SLOC, especially during the early stages of development. An alternative method, **function points,** should be used to estimate software size. Function points are used primarily for MISs, whereas, **feature points** (similar to function points) are used for real-time or embedded systems. [PUTNAM92] SLOC and function (and feature) points are valuable size estimation techniques. Table 8-5 summarizes the differences between the function point and SLOC methods.

FUNCTION POINTS	SOURCE LINES-OF-CODE
Specification-based	Analogy-based
Language independent	Language dependent
User-oriented	Design-oriented
Variations a function of counting conventions	Variations a function of languages
Expandable to source lines-of-code	Convertable to function points

Table 8-5 Function Points versus Lines-of-code

Source Lines-of-Code Estimates

Most **source lines-of-code (SLOC)** estimates count all executable instructions and data declarations but exclude comments, blanks, and continuation lines. SLOC can be used to estimate size through analogy — by comparing the new software's functionality to similar functionality found in other historic applications. Obviously, having more detailed information available about the functionality of the new software provides the basis for a better comparison. In theory, this should yield a more credible estimate. The relative simplicity of the SLOC size measure facilitates automated and consistent (repeatable) counting of actual completed software size, as well as storing and retrieving the size data needed to prepare an accurate estimate for future efforts. The most significant advantage of SLOC estimates is that they directly relate to the software to be built. The software can then be measured after the fact and compared with your initial estimates. [HUMPHREY89] If you are using SLOC with a predictive model (e.g., COCOMO), your estimates will need to be continually updated as new information is available. Only through this constant re-evaluation can the predictive model provide a cost that approximates actuals.

A large body of literature and historical data exists that uses SLOC, or thousands of source lines-of-code (KSLOC), as the size measure. Source lines-of-code are easy to count and most existing software estimating models use SLOCs as the key input. However, it is virtually impossible to estimate SLOC from initial requirements statements. Their use in estimation requires a level of detail that is hard to achieve (i.e., the planner must estimate the SLOC to be produced before sufficient detail is available to accurately do so.) [PRESSMAN92]

Because SLOCs are language-specific, the definition of how SLOCs are counted has been troublesome to standardize. This makes comparisons of size estimates between applications written in different programming languages difficult although conversion factors are available. From SLOC estimates a set of simple, size-oriented productivity and quality metrics can be developed for any given on-going program. These metrics can be further refined using productivity and quality equations such as those found in the basic COCOMO model.

Function Point Size Estimates

Function points, as defined by A.J. Albrecht, are the weighted sums of five different factors that relate to user requirements:

- Inputs,
- Outputs,
- Logic (or master) files,
- Inquiries, and
- Interfaces. [ALBRECHT79]

The **International Function Point Users Group (IFPUG)** is the focal point for function point definitions. The basic definition of function points provided above has been expanded by several others to include additional types of software functionality, such as those related to embedded weapons systems software (i.e., feature points).

Function points are counted by first tallying the number of each type of function, as listed above. These unadjusted function point totals are subsequently adjusted by applying **complexity measures** to each type of function point. The sum of the total complexity-adjusted function points (for all types of function points) becomes the total adjusted function point count. Based on prior experience, the final function point figure can be converted into a reasonably good estimate of required development resources. *[For more information on function point counting, see the "Counting Practices Manual" available from the IFPUG administrative office in Westerville, Ohio for a nominal charge, (614) 895-3170 or Fax (614) 895-3466.]*

Table 8-6 illustrates a function point analysis for a nominal program. First you count the number of inputs, outputs, inquiries, logic files, and interfaces required. These counts are then multiplied by established values. The total of these products is adjusted by the degree of

	Simple	Average	Complex	Total
Inputs	3X __	4X 2	6X 2	20
Outputs	4X 1	5X 3	7X __	19
Inquiries	3X __	4X __	6X __	0
Files	7X __	10X 1	15X __	10
Interfaces	5X __	7X __	10X 1	10
UNADJUSTED FUNCTION POINTS = 59				

Table 8-6 Function Point Computation [REIFER92]

complexity based on the estimator's judgment of the software's complexity. **Complexity judgments** are domain-specific and include factors such as data communications, distributed data processing, performance, transaction rate, on-line data entry, end-user efficiency, reusability, ease of installation, operation, change, or multiple site use. This process for our nominal program is illustrated in Figure 8-11.

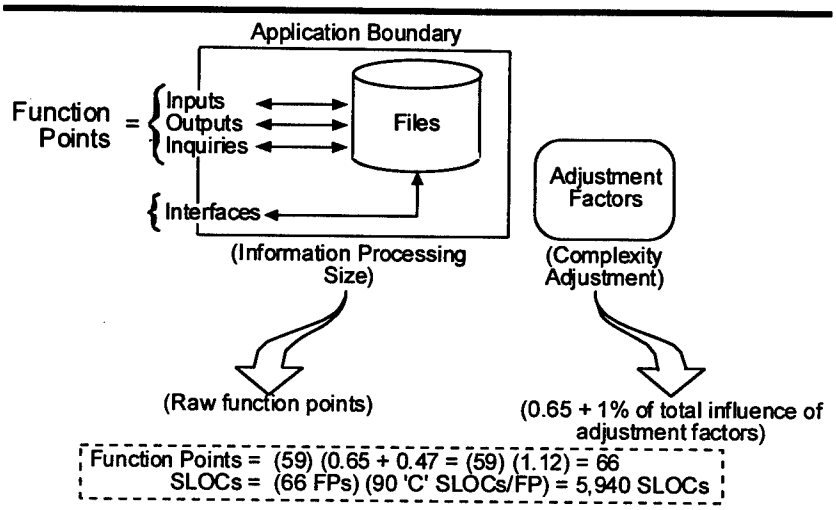


Figure 8-11 Function Point Software Size Computational Process [REIFER92]

While function points aid software size estimates, they too have drawbacks. At the very early stages of system development, function points are also difficult to estimate. Additionally, the complexity factors applied to the equation are subjective since they are based on the analyst/engineer's judgment. Few automated tools are available to count either unadjusted or adjusted function points, making comparisons between or among programs difficult, and making the function point counts for any single program inconsistent when calculated by different analysts. However, function points are valuable in making early estimates, especially after the SRS has been completed. Like SLOC, they too are affected by changes in system and/or software requirements. Also, as a relatively new measure of software size, there are few significant, widely-available databases for estimating function points by comparison (analogy) to functionally similar historic software applications.

Feature Point Size Estimates

A derivative of function points, **feature points** were developed to estimate/measure real-time systems software with high algorithmic complexity and generally less inputs/outputs than MISs. **Algorithms** are sets of mathematical rules expressed to solve significant computational problems. For example, a square root extraction routine, or a Julian date conversion routine, are algorithms.

In addition to the five standard function point parameters, feature points include an algorithm(s) parameter which is assigned the default weight of 3. The feature point method reduces the empirical weights for logical data files from a value of 10 to 7 to account for the reduced significance of logical files in real-time systems. For applications in which the number of algorithms and logical data files are the same, function and feature point counts generate the same numeric values. But, when there are more algorithms than files, feature points produce a greater total than function points. Table 8-7 illustrates the ratio of function point to feature point counts for selected applications. *[For a more detailed explanation of feature points, see Capers Jones, Applied Software Measurement.]* [JONES91]

APPLICATION	FUNCTION POINTS	FEATURE POINTS
Batch MIS projects	1	0.80
On-line MIS projects	1	1.00
On-line database projects	1	1.00
Swtiching systems projects	1	1.00
Embedded real-time projects	1	1.35
Factory automation projects	1	1.50
Diagnostic and prediction projects	1	1.75

Table 8-7 Ratios of Feature Points to Function Points [JONES91]

NOTE: See Volume 2, Appendix J, *SPR Metric Analysis: Counting Rules for Function Points and Feature Points.*

Complexity

Complexity measures focus on designs and actual code. They assume there is a direct correlation between design complexity and design errors, and code complexity and latent defects. By recognizing the properties of each that correlate to their complexity, we can identify those high-risk applications that either should be revised or subjected to additional testing.

Those software properties which correlate to how complex it is are **size, interfaces among modules** (usually measured as *fan-in*, the number of modules invoking a given application, or *fan-out*, the number of modules invoked by a given application), and **structure** (the number of paths within a module). Complexity metrics help determine the number and type of tests needed to cover the design (interfaces or calls) or coded logic (branches and statements).

There are several accepted methods for measuring complexity, most of which can be calculated by using automated tools. Addendum C, by McCabe and Watson, discusses the **McCabe Cyclomatic Complexity Metric**. The **Halstead Volume Metric** (also mentioned in Addendum B) identifies certain intrinsic, measurable properties embodied in an algorithm. Halstead's theory is based on the assumption that "*the human brain follows a more rigid set of rules (in developing algorithms) than it has been aware of...*" [HALSTEAD77] He defines a set of primitive measures that can be derived estimated once the design is complete, after the software is coded. These are:

- n_1 = the number of distinct operators used,
- n_2 = the number of distinct operands used,
- N_1 = the total number of operators used, and
- N_1 = the total number of operands used.

Halstead uses these measures to derive expressions for total application *length*, the potential minimum algorithm *volume*, the *actual volume* (number of bits need to specify an application), the *program level* (a software complexity measure), the *language level* (constant for any given language), and other features such as development *effort*, development *time*, and projected number of software *defects*. [PRESSMAN92]

To get an idea of how complexity affects software quality (in this case latent defects), the **Card Design Complexity** metric is an interesting

example. In his book, Measuring Software Design Quality, Card defines a design complexity measure he calls "C_T." Card performed a number of studies to develop this overall design metric based on a composite calculated measure. The formula for Card Design Complexity metric is:

$$C_T = \frac{S_T}{n} + \frac{D_T}{n}$$

where,

C_T = the sum of the fan-out squared (over each module in the design),

D_T = the sum of the number of module input and output variables divided by fan-out + 1 (over each module in the design), and

n = the number of modules in the application.

Card conducted an independent validation study on data taken from eight programs (about 2,000 modules) and found a correlation between C_T and defect density (defects KSLOC) to be .83. His study evaluated coupling and cohesion as predictors (metrics) for defects and fault rate. As you remember from Chapter 4, *Engineering Software-Intensive Systems*, ***coupling*** is the measure of interface tightness among modules, and ***cohesion*** is the measure of how tightly bound or related internal module elements are to one another. He classified several hundred modules for which defect data were available into three groups based on their coupling rating: **parameter coupling** (low complexity), **mixed coupling** (medium complexity), and **extensive coupling** (high complexity). Based on their cohesion strength he classified them as: only one function **(low complexity)**, two functions (**medium complexity**), and three or more functions (**high complexity**). The analysis of the defects and fault rates for each of these groupings is illustrated on Figure 8-12 (below).

As you can see, Card found the relationship between coupling and defect rate to be inconsequential; whereas, the correlation between cohesion (functional robustness) and defect rate was quite significant. The highly robust modules had a higher percent of zero faults (50% versus 18%) and a lower percent with high faults (20% versus 44%) than the highly complex modules. This is an example of why code complexity must be measured and controlled throughout the life cycle. [CARD90]

NOTE: See Addendum C, *Software Complexity*, by Thomas McCabe for a discussion on complexity analysis.

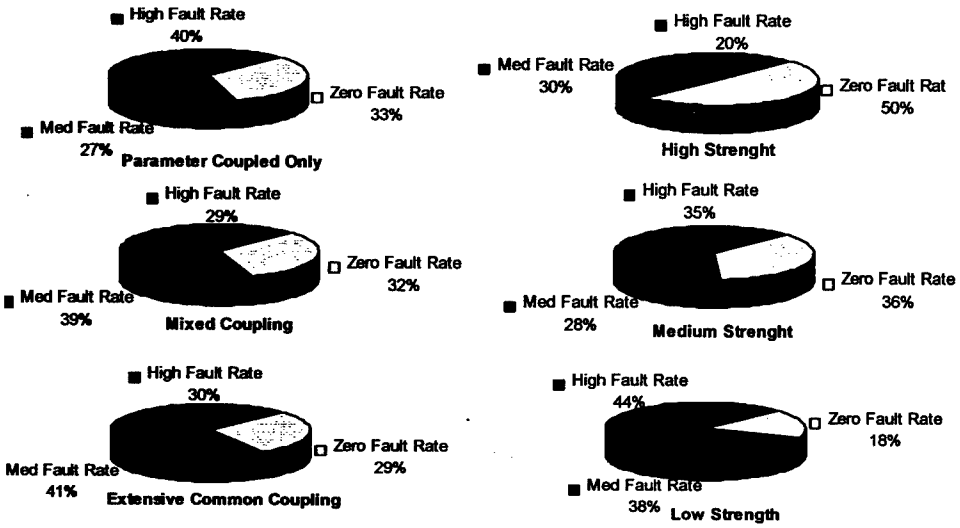


Figure 8-12 Coupling and Cohesion versus Fault Rate [HETZEL93]

Requirements

As you learned in Chapter 1, *Software Acquisition Overview*, requirements changes are a major source of software size risk. If not controlled and baselined, ***requirements creep*** is a common problem that increases cost, schedule, and fielded defects. If requirements evolve as the software evolves, it is next to impossible to develop a successful product. Software developers find themselves shooting at a moving target and throwing away design and code faster than they can crank it out *[scrap and rework is discussed below]*. **Colonel Robert Lyons, Jr.**, former co-leader of the F-22 System Program Office Avionics Group, cites an *"undisciplined requirements baselining process"* as the number one cause of *"recurring weapons system problems."* An undisciplined requirements process is characterized by:

- Inadequate requirements definition,
- Late requirements clarification,
- Derived requirements changes,
- Requirements creep, and
- Requirements baselined late. [LYONS91]

Once requirements have been defined, analyzed, and written into the System Requirements Specification (SRS), they must be tracked throughout subsequent phases of development. In a system of any

size this is a major undertaking. The design process translates user-specified (or **explicit**) requirements into derived (or **implicit**) requirements necessary for the solution to be turned into code. This multiplies requirements by a factor of sometimes hundreds. [GLASS92]

Each implicit requirement must be fulfilled, traced back to an explicit requirement, and addressed in design and test planning. ***It is the job of the configuration manager to guarantee the final system meets original user requirements.*** As the requirements for a software solution evolve into a design, there is a snowball effect when converting original requirements into design requirements needed to convert the design into code. Conversely, sometimes requirements are not flowed down and get lost during the development process *(dropped through the developmental crack)* with a resulting loss in system performance or function. When requirements are not adequately tracked, interface data elements can disappear, or extra interface requirements can be introduced. Missing requirements may not become apparent until system integration testing, where the cost to correct this problem is exponentially high.

Effort

In the 1970s, **Rome Air Development Center (RADC)** collected data on a diverse set of over 500 DoD programs. The programs ranged from large (millions of lines-of-code and thousands of months of effort) to very small (a one month effort). The data was sparse and rather primitive but it did include output KLOC and input effort months and duration. At that time most program managers viewed effort and duration as interchangeable. When we wanted to cut completion time in half, we assigned twice as many people! Lessons-learned hard knocks and program measures such as the RADC database, indicated that the relationships between duration and effort were quite complex, nonlinear functions. Many empirical studies over the years have shown that manpower in large developments builds up in a characteristic way and that it is complex power function of software size and duration. Many estimation models were introduced, the best known of which is **Barry Boehm**'s **COnstructive COst MOdel (COCOMO).** [HETZEL93]

Productivity

Software productivity is measured in the number of lines-of-code or function/feature points delivered (i.e., SLOC that have been produced, tested, and documented) per staff month that result in an acceptable and usable system. Table 8-8 lists the industry average software productivity rates (in function points produced) over 5-year intervals.

Year	System Software	Commercial Software	Information Software	Military Software	Overall Average
1945	--	--	--	0.1	0.1
1950	0.5	--	1.0	0.3	0.5
1955	0.6	--	1.5	0.4	0.8
1960	1.0	1.5	2.0	0.5	1.3
1965	1.5	2.5	3.0	0.7	1.9
1970	2.0	3.5	4.0	1.0	2.6
1975	2.5	4.5	5.0	1.5	3.4
1980	3.0	5.0	6.0	2.0	4.0
1985	3.5	6.0	7.0	2.5	4.9
1990	4.0	7.5	8.0	3.0	5.6
1995*	6.0	10.5	12.5	4.0	8.3
2000*	6.5	15.0	18.0	6.5	11.5

Copyright© 1993 by SPR. All rights reserved.

Table 8-8 Industry Average Productivity Rates (function points produced per staff month)

Boehm explains there are three basic ways to improve **software development productivity**.

- Reduce the cost-driver multipliers,
- Reduce the amount of code; and,
- Reduce the scalable factor that relates the number of instructions to the number of manmonths or dollars.

His **model for measuring productivity** is:

$$Effort = Constant \times Size^{Sigma} \times Multipliers$$

[BOEHM89]

In this equation, multipliers are factors (such as efficiency of support tools, whether the software must perform within limited hardware constraints, personnel experience and skills, etc.). Figure 8-13 lists the various cost drivers that affect software development costs. Many of these factors (not all) can be modified by effective management practices. The weight of each factor as a **cost multiplier** (on a scale of 1 to 5, with 5 having the greatest weight) reflects the relative affect that factor has on total development costs. Boehm's studies show that next to size, ***"employing the right people" has the greatest influence on productivity***. (Reliability and complexity are also important multipliers.)

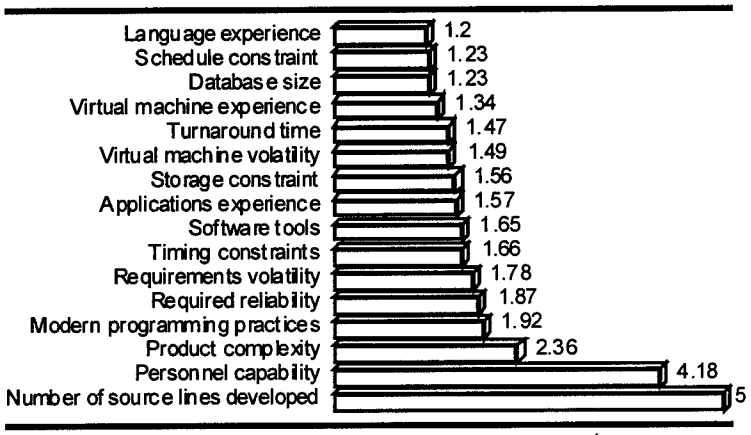


Figure 8-13 Software Productivity Factors (Weighted Cost Multipliers) [BOEHM89]

People with more Ada experience build better Ada software! By building better software, *"Ada can reduce the added cost of building highly complex, highly reliable software."* [BOEHM89] Advanced tools and environments reduce turnaround times and defect rates, which reduce the numerical value of the multipliers. Reducing the size can be accomplished by using reusable code, COTS software, or VHLLs (which will some day produce the same function in very few instructions).

The third element in the equation, the exponent *Sigma*, is about 1.2 in **Boehm's COCOMO model**. For large aerospace systems the value amounts to a fairly costly exponent. When you double the size of software, you multiply the cost by $2^{1.2}$, which is 2.3. In other words, the cost is doubled, plus a 30% penalty for size. The size penalty, according to Boehm, results from inefficiency influences that are a product of size. The bigger the software development, the bigger the integration issues, the bigger the team you need, the less efficient they become.

> *You bring a bigger team on board. The people on the team spend more time talking to each other. There is more learning curve effect as you bring in people who don't know what you are building. There is a lot of thrashing in the process any time there is a proposed change of things that haven't been determined that people are trying to determine. Any time you do have a change, there are ripple effects that are more inefficient on big programs than on small programs.* [BOEHM89]

Cost and Schedule

As previously discussed, the driving factors in DoD software development have always been cost, schedule, or both. A typical DoD scenario has been for the software development schedule to be accelerated to support the overall program schedule, increasing the cost of the software and reducing quality. Because ***cost is the key issue in any development program***, it must be reviewed carefully as part of the program review and approval process. As Benjamin Franklin explained:

> *I conceive that the great part of the miseries of mankind are brought upon them by the false estimates they have made of the value of things.* [FRANKLIN33]

To avoid the miseries of a runaway program, ***you must carefully plan for and control the cost and schedule of your software development effort.*** These measures are important for determining and justifying the required funding, to determine if a specific proposal is reasonable, and to insure that the software development schedule is consistent with the overall system schedule. Cost measures should also be used to evaluate whether the developer has the appropriate mix and quantity of assigned staff, and to develop a reasonable

program cost and schedule baseline for effective and meaningful program management.

While size is by far the most significant driver of cost and schedule, other factors impact them as well. These factors are usually more qualitative in nature and address the development and operational environments as well as the software's characteristics. Most software cost estimating models use these factors to determine environmental and complexity factors which are in turn used in computations to calculate effort and cost.

Multisource cost and **schedule estimation** is the use of multiple, independent organizations, techniques, and models to estimate cost and schedule, including analysis and iteration of the differences between estimates. Whenever possible, multiple sources should be used for estimating any unknowns, not just cost and schedule. Errors or omissions in estimates can often be identified by comparing one with another. Comparative estimates also provide a sounder set of *"should-costs"* upon which to control software development. As with size estimates, assessment from alternate sources (such as program office software technical staff, prime or subcontractors, or professional consulting firms is advisable for cost and schedule. Reassessments throughout the program life cycle improve the quality of estimates as requirements become better understood and refined. The following summarizes the resources you should consider when costing software development.

- **Human resources**. This includes the number and qualifications of the people required, as well as their functional specialties. Boehm asserts that ***human resources are the most significant cost drivers on a software development effort.*** [BOEHM81] Development personnel skills and experience (reflected in their productivity) have the greatest effect on cost and schedule. *[See Chapter 9 for a detailed discussion on productivity cost drivers.]*
- **Hardware resources**. This includes development (host) and target computers, and compilers. Hardware resources used to be major cost drivers when development personnel needed to share equipment with multiple constituencies. Now that virtually everyone has a PC or workstation on their desk, the issue is whether the target computer significantly differs from the development computer. For instance, if the target machine is an air or spaceborne system, the actual CPU may be technology-driven and not usable for all required development activities.

- **Software resources**. Software is also used as a tool to develop other software. CASE tools needed for development, test, and code generation must be considered. As discussed in Chapter 10, *Software Tools*, your toolset might include: business systems planning tools, program management tools, support tools, analysis and design tools, programming tools, integration and test tools, prototyping and simulation tools, maintenance tools, cost/schedule estimating tools, and architectural tools.
- **Reusable resources**. Reusable assets *[defined in Chapter 9, Reuse]* are a valuable resource that must be considered in determining your cost requirements. This includes the assets you will develop for future reuse by other programs, as well as searching the reuse repositories for existing code that can be integrated into your development. Reusable assets will have significant impact on your program cost and schedule.

Schedule measurements track the contractor's performance towards meeting commitments, dates, and milestones. **Milestone performance metrics** give you a graphical portrayal (data plots and graphs) of program activities and planned delivery dates. It is essential that what constitutes progress slippage and revisions is understood and agreed upon by both the developer and the Government. Therefore, entry and exit criteria for each event or activity must be agreed upon at contract award. A caution in interpreting schedule metrics is to keep in mind that many activities occur simultaneously. Slips in one or more activities usually impact on others. Look for problems in process and ***never, never sacrifice quality for schedule!***

NOTE: See Chapter 15, *Managing Process Improvement*, for a discussion on the Cost/Schedule Control System Criteria (C/SCSC) method for measuring, tracking, and analyzing contract performance earned-value.

Cost and Schedule Estimation Methodologies/ Techniques

Most estimating methodologies are predicated on analogous software programs. **Expert opinion** is based on experience from similar programs; **parametric models** *[discussed in Chapter 10, Software Tools]* stratify internal data bases to simulate environments from many analogous programs; **engineering builds** reference similar experience at the unit level; and **cost estimating relationships** (like parametric models) regress algorithms from several analogous

programs. Deciding which of these methodologies (or combination of methodologies) is the most appropriate for your program usually depends on ***availability of data***, which is in turn depends on where you are in the life cycle or your scope definition.

- **Analogies.** Cost and schedule are determined based on data from completed similar efforts. When applying this method, it is often difficult to find analogous efforts at the total system level. It may be possible, however, to find analogous efforts at the subsystem or lower level (CSCI/CSC/CSU). Furthermore, you may be able to find completed efforts similar more or less in complexity. If this is the case, a *scaling factor* may be applied based on expert opinion (e.g., CSCI-x is 80% as complex). After an analogous effort has been found, associated data need to be assessed. It is preferable to use effort rather than cost data; however, if only cost data are available, these costs must be normalized to the same base year as your effort using current and appropriate inflation indices. As with all methods, the quality of the estimate is directly proportional to the credibility of the data.
- **Expert (engineering) opinion**. Cost and schedule are estimated by determining required effort based on input from personnel with expansive experience on similar programs. Due to the inherent subjectivity of this method, it is especially important that input from several independent sources be used. It is also important to request only effort data rather than cost data as cost estimation is usually out of the realm of engineering expertise (and probably dependent on non-similar contracting situations). This method is rarely used as a primary methodology alone, with the exception of rough orders-of-magnitude estimates. Expert opinion is used to estimate lower-level, low cost, pieces of a larger cost element when a labor-intensive cost estimate is not feasible.
- **Parametric models**. See Chapter 10, *Software Tools*, for a discussion on parametric models for cost and schedule estimation.
- **Engineering build** (*grass roots*, or *bottoms-up* build). Cost and schedule are determined by estimating effort based on the effort summation of detailed functional breakouts of tasks at the lowest feasible level of work. For software, this requires a detailed understanding of the software architecture. Analysis is performed at the CSC or CSU level and associated effort is predicted based on unit level comparisons to similar units. Often, this method is based on a notional system of *government estimates of most probable cost* and used in source selections before contractor solutions are known. This method is labor-intensive and is usually performed with engineering support; however, it provides better assurance

than other methods that the entire development scope is captured in the resulting estimate.

- **Cost Performance Report (CPR) analysis**. Future cost and schedule estimates are based on current progress. This method may not be an optimal choice for predicting software cost and schedule because software is generally developed in three distinct phases (requirements/design, code/unit test, integration/test) by different teams. Apparent progress in one phase may not be predictive of progress in the next phases, and lack of progress in one phase may not show up until subsequent phases. For example, it is difficult to measure *earned-value [discussed in Chapter 15, Managing Process Improvement]* during the requirements phase because it depends on counting completed documents at scheduled milestones. Difficulty in implementing a poor design may occur without warning, or problems in testing may be the result of poor test planning or previously undetected coding defects. CPR analysis can be a good starting point for identifying problem areas, and problem reports included with CPRs may provide insight for risk assessments.
- **Cost estimating relationships (CERs)/factors**. Cost and schedule are estimated by determining effort based on algebraic relationships between a dependent (effort or cost) variable and independent variables. This method ranges from using simple factor, such as cost per line-of-code on similar program with similar contractors, to detailed multi-variant regressions based on several similar programs with more than one causal (independent) variable. Statistical packages are commercially available for developing CERs, and if data are available from several completed similar programs (which is not often the case), this method may be a worthwhile investment for current and future cost and schedule estimating tasks. Parametric model developers incorporate a series of CERs into an automated process by which parametric inputs determine which CERs are appropriate for the program at hand.

Of these techniques, the most commonly used is parametric modeling. There currently is no list of recommended or approved models; however, you will need to justify the appropriateness of the specific model or other technique you use in an estimate presented for DAB and/or MAISARC Review. As mentioned above, determining which method is most appropriate is driven by the availability of data. Regardless of which method used, a thorough understanding of your software's functionality, architecture, and characteristics, and your contract is necessary to accurately estimate required effort, schedule, and cost.

NOTE: Refer to "*A Manager's Checklist for Validating Software Cost and Schedule Estimates*," CMU/SEI-95-SR-04, and "*Checklists and Criteria for Evaluating the Cost and Schedule Estimating Capabilities of Software Organizations*," CMU/SEI-95-SR-05.

Ada-Specific Cost Estimation

Using Ada-specific models is necessary because Ada developments do not follow the classic patterns included in most traditional cost models. As stated above, the time and effort required during the design phase are significantly greater (50% for Ada as opposed to 20% for non-Ada software developments). [Ada/C++91] Another anomaly with Ada developments is **productivity rates**. Traditional non-Ada developments have historically recorded that productivity rates decrease as program size increases. With Ada, the opposite is often true. Due in large to Ada reusability, the larger the program size — the greater the productivity rate. Ada estimating models and training sources are listed in Volume 2, Appendices A and B.

Scrap and Rework

A major factor in both software development cost and schedule is that which is either **scrapped** or **reworked**. The **costs of conformance** are the normal costs of preventing defects or other conditions that may result in the scrapping or reworking of the software. The **costs of nonconformance** are those costs associated with redoing a task due to the introduction of an error, defect, or failure on initial execution (including costs associated with fixing failures that occur after the system is operational, i.e., scrap and rework cost).

NOTE: Good planning requires consideration of the *"rework cycle."* For iterative development efforts, *rework can account for the majority of program work content and cost!*

Rework costs are very high. ***Boehm's data suggest rework costs are about 40% of all software development expenditures.*** **Defects** that result in rework are one of the most significant sources of risk in terms of cost, delays, and performance. You must encourage and demand that your software developer effectively measures and controls defects. Rework risk can be controlled by:

- Using procedures to identify defects as early as possible;
- Examining the root causes of defects and introducing process improvements to reduce or eliminate future defects; and
- Developing incentives that reward contractors/developers for early and comprehensive defect detection and removal. *[Defect causal analysis, detection, removal, and prevention are discussed in Chapter 15, Managing Process Improvement.]*

There are currently no cost estimating models available that calculate this substantial cost factor. However, program managers must measure and collect the costs associated with software scrap and rework throughout development. First, it makes good sense to monitor and track the cost of defects, and thereby to incentivize closer attention to front-end planning, design, and other defect preventive measures. Second, by collecting these costs across all software development programs, parametric models can be designed to better help us plan for and assess the acquisition costs associated with this significant problem.

NOTE: See Volume 2, Appendix O, Chapter 8 Addendum D, *Swords and Plowshares; The Rework Cycles of Defense and Commercial Software Development Projects."*

Support

Software supportability progress can be measured by tracking certain key supportability characteristics. With these measures, both the developer and the acquirer obtain knowledge which can be focused to control supportability.

- **Memory size.** This metric tracks spare memory over time. The spare memory percentage should not go below the specification requirement.
- **Input/output.** This metric tracks the amount of spare I/O capacity as a function of time. The capacity should not go below the specification requirement.
- **Throughput**. This metric tracks the amount of throughput capacity as a function of time. The capacity should not go below specification requirements.
- **Average module size.** This metric tracks the average module size as a function of time. The module size should not exceed the specification requirement.

- **Module complexity.** This metric tracks the average complexity figure over time. The average complexity should not exceed the specification requirement.
- **Error rate.** This metric tracks the number of errors compared to number of errors corrected over time. The difference between the two is the number of errors still open over time. This metric can be used as a value for tested software reliability in the environment for which it was designed.
- **Supportability.** This metric tracks the average time required to correct a deficiency over time. The measure should either remain constant or the average time should decrease. A decreasing average time indicates supportability improvement.
- **Lines-of-code changed.** This metric tracks the average lines-of-code changed per deficiency corrected when measured over time. The number should remain constant to show the complexity is not increasing and that ease of change is not being degraded.

NOTE: See Volume 2, Appendix O, Chapter 8 Addendum D, *"Making Metrics Work Miracles."*

CAUTIONS ABOUT METRICS

Software measures are valuable for gaining insight into software development; however, they are not a solution to issues in and of themselves. To implement a metrics program effectively, you must be aware of certain limitations and constraints.

- **Metrics must be used as indicators, not as absolutes.** Metrics should be used to prompt additional questions and assessments not necessarily apparent from the measures themselves. For instance, you may want to know why the staff level is below what was planned. Perhaps there is some underlying problem, or perhaps original manpower estimates need adjusting. Metrics cannot be applied in a vacuum, but must be combined with program knowledge to reach correct conclusions.
- **Metrics are only as good as the data that support them.** Input data must be timely, consistent, and accurate. A deficiency in any of these areas can skew the metrics derived from the data and lead to false conclusions.
- **Metrics must be understood to be of value.** This means understanding what the low-level measurement data represent and how they relate to the overall development process. You must look

beyond the data and measurement process to understand what is really going on. For example, if there is a sharp decrease in **defect detection** and an increase in defect resolution and close out, you might conclude that the number of inserted defects is decreasing. However, in a resource-constrained environment, the defect discovery rate may have dropped because engineering resources were temporarily moved from defect detection (e.g., testing) to defect correction.

- **Metrics should not be used to judge your contractor (or individual) performance.** Measurement requires a team effort. While it is necessary to impose contractual provisions to implement software measurement, it is important not to make metrics a controversial issue between you and your contractor. ***Support of the measurement process will be jeopardized if you "shoot-the-messenger."*** Measurements should be used to identify problem areas and for improving the process and product. While metrics may deal with personnel and organizational data, these data must be used for constructive, process-oriented decision-making, rather than for placing blame on individuals or teams.
- **Metrics cannot identify, explain, or predict everything.** Metrics must be used in concert with sound, hands-on management practice. They are only valuable if used to augment and enhance intimate process knowledge and understanding.
- **Analysis of metrics should NOT be performed exclusively by the contractor.** Ideally, the contractor you select will already have a metrics process in place. As mentioned above, you should implement your own independent metrics analysis process because:
  - Metrics analysis is an iterative process reflecting issues and problems that vary throughout the development cycle;
  - The natural tendency of contractors is to present the program in the best light; therefore, independent government analysis of the data is necessary to avoid misrepresentation; and
  - Metrics analysis must be issue-driven and the government and contractor have inherently different issue perspectives.
- **Direct comparisons of programs should be avoided.** No two programs are alike; therefore, any historical data must be tailored to your program specifics to derive meaningful projections. ***[Conversely, do not tailor your data to match historical data.]*** However, metrics from other programs should be used as a means to establish *normative values* for analysis purposes.
- **A single metric should not be used.** No single metric can provide the insight needed to address all program issues. Most issues require multiple data items to be sufficiently characterized.

Because metrics are interrelated, you must correlate trends across multiple metrics. [ROZUM92]

NOTE: See Volume 2, Appendix O, Chapter 8 Addendum C, *"Metrics: The Measures of Success."*

ADDRESSING MEASUREMENT IN THE RFP

Your RFP should define the indicators and metrics the Government needs to track progress, quality, schedule, cost, and maintainability. What you should look for when analyzing an offeror's **Metrics Usage Plan** is ***"control."*** Through measurement, the process's internal workings are defined and assessed. If an effective process improvement plan is executed (which requires appropriate measurements be taken) data are collected and analyzed to predict process failures. Therefore, the offeror must have a corporate mechanism implemented in a systematic manner that performs orderly process control and methodical process improvement. This can be identified by the **measurement methods** the company uses to assess the development process, analyze the data collected, and feed back corrections for problems within the process. [CAREY92]

Make sure the software quality metrics and indicators they employ include a clear definition of component parts (e.g., SLOC), are accurate and readily collectible, and span the development spectrum and functional activities. They must identify metrics early and apply them at the beginning of the system engineering and software implementation process. They should also develop a software Metrics Usage Plan before contract award.

REFERENCES

[Ada/C++91] Ada and C++ Business Case Analysis, Deputy Assistant Secretary of the Air Force (Communications, Computers, and Logistics), Washington, DC, July 1991

[ALBRECHT79] Albrecht, A.J., "Measuring Application Development Productivity," *Proceedings of the IBM Applications Development Symposium*, Monterey, California, October 1979

[BOEHM81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981

[BOEHM89] Boehm, Barry W., as quoted by Ware Myers, "Software Pivotal to Strategic Defense," *IEEE Computer*, January 1989

[CAMPBELL95] Campbell, Luke and Brian Koster, "Software Metrics: Adding Engineering Rigor to a Currently Ephemeral Process," briefing presented to the McGrummwell F/A-24 CDR course, 1995

[CARD90] Card, David N., and Robert L. Glass, Measuring Software Design Quality, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1990

[CAREY92] Carey, Dave and Don Freeman, "Quality Measurements in Software," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[DeMARCO86] DeMarco, Tom, Controlling Software Projects, Yourdon Press, New York, 1986

[FRANKLIN33] Franklin, Benjamin, Poor Richard's Almanac, 1733

[GILLIGAN94] Gilligan, John, "The C4I Acquisition Environment: Perspectives and Challenges," briefing presented at the ISAC meeting, November 21, 1994

[HALSTEAD77] Halstead, M.H., Elements of Software Science, North Holland, Amsterdam, 1977

[HETZEL93] Hetzel, Bill, Making Software Measurement Work: Building an Effective Measurement Program, QED Publishing Group, Boston, 1993

[HUMPHREY89] Humphrey, Watts S., Managing the Software Process, The SEI Series in Software Engineering, Addison-Wesley Publishing Company, Inc., 1989

[IEEE83] ANSI/IEEE Standard 729-1983, *IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers, Inc., New York, 1983

[JONES91] Jones, Capers, Applied Software Measurement, McGraw-Hill, New York, 1991

[LYONS91] Lyons, Lt Col Robert P., Jr., "Acquisition Perspectives: F-22 Advanced Tactical Fighter," briefing presented to Boldstroke Senior Executive Forum on Software Management, October 16, 1991

[MARCINIAK90] Marciniak, John J. and Donald J. Reifer, Software Acquisition Management: Managing the Acquisition of Custom Software Systems, John Wiley & Sons, Inc., New York, 1990

[PEHRSON96] Pehrson, Ronald J., "Software Development for the Boeing 777," *CrossTalk*, January 1996

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[PUTNAM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software On Time, Within Budget, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992

[ROZUM92] Rozum, James A., *Software Measurement Concepts for Acquisition Program Managers*, Technical Report CMU/SEI-92-TR-11/ ESD-TR-92-11, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, June 1992

[YOURDON92] Yourdon, Edward, Decline and Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992

Blank page.

CHAPTER 8
Addendum A

Assessment Metrics for Use with the Capability Maturity Model: Are We Improving?

S.J. Leadabrand
W.E. Burns
Loral Vought Systems Corporation

The Software Engineering Institute (SEI) Software Process Assessment (SPA) is a mechanism that measures "*where we are*" relative to the SEI Capability Maturity Model (CMM). A standard questionnaire is completed for each of the projects included in the SPA. Subsequent interviews and analysis are used to adjust and correct the questionnaire results. A determination is then made as to the "*maturity level*" of the total organization being assessed.

There is additional useful information that can be obtained through this process by computing metrics using the same data that has already been collected. These metrics can be used to further analyze the state of the organization with respect to the strengths, weaknesses and consistency of its practices. They can provide both the Software Engineering Process Group (SEPG) and management with baselines from which to measure improvement.

"ORGANIZATIONAL PROFILE" METRIC

The first metric is the "*organizational profile*" as shown in Figure 8-14. The questions pertaining to each maturity level are separated to produce four profiles from Level 2 through Level 5. The profile (P) for each project (j) is produced by adding the number of "*yes*" answers for each question from the questionnaire. This sum is then divided by the adjusted total number of projects. The adjusted total is computed by subtracting the number of "*not applicable*" answers from the total number of answers to the question. The result is then converted to a percentage.

$$Pj = \frac{\sum_{i=1}^{I} Y_{ij}}{I - \sum_{I=1}^{I} NA_{ij}}$$

where
Y = 1 if the project answered "*yes*" to the question;
i = the project sequence number (i = 1,2, ... I), where
I = the total number of projects assessed;
j = the j_{th} question within the level (j = 1,2, ... **J**) out of **J**, where
J = the total number of questions; and
NA = I if the project answered "*not applicable*" to the question.

This metric is easily computed in a spreadsheet with the results available for output in bar graph style.

Figure 8-14 aids in the interpretation of the consistency of application of the practices. In this fictitious example, it can be seen that the organization has three or four areas of strength indicated by those bars at 80% or better. All of the practices inferred by the question set are being practiced at least on some of the projects. Of particular interest is the number of practices that are inconsistent across the projects. These are indicated by percentages in the mid-range, e.g., 35 - 70%, on the graph. These "*organizational profile*" metrics are of particular interest to the SEPG since they can be used as a baseline from which improvement in specific areas can be measured. They are probably not of general interest since they include so much detail.

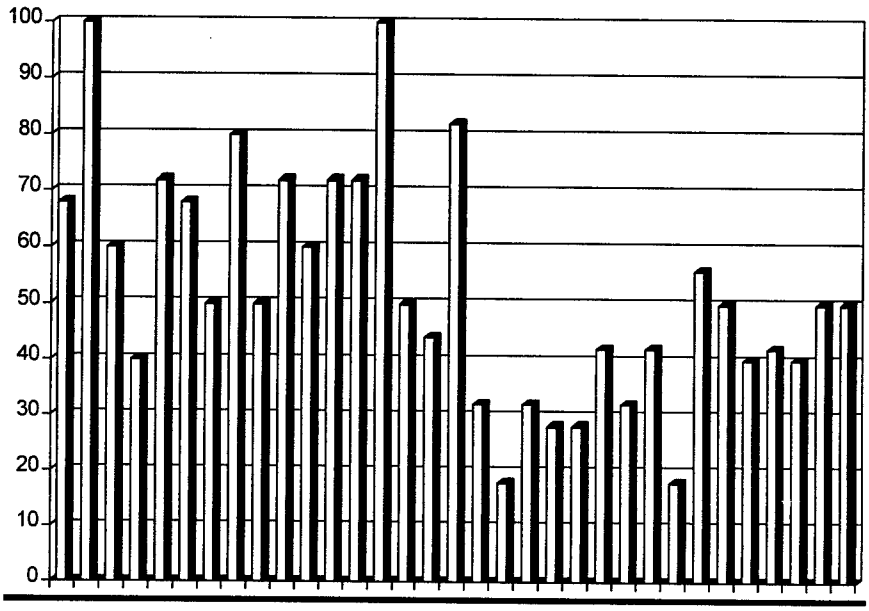


Figure 8-14 Normalized Software Process Questionnaire Results, Level 2

"WHERE WE ARE" METRIC

A metric of more general interest is the "*where we are*" metric (s_i) shown in Figure 8-15 (below). This metric is computed separately for each maturity level of the CMM. One of the most frequently asked questions is, *"How close to the next maturity level are we?"* Another is, "*How much progress are we making?*" This metric can help answer both of these questions. It is suitable for presentation to management since it gives an overall "*flavor*" view without getting bogged down in the details.

The "*where we are*" metrics are also separated into two categories: "*all the questions*" and the "*asterisked questions*." The computation consists of summing the "*yes*" answers in each of the two categories for those questions applicable to each maturity level. The sum is adjusted for not-applicable answers and the result is converted to a percentage. This percentage can be compared against the SEI's threshold for that maturity level.

$$Si = \frac{\sum_{j=1}^{j} Y_{ij}}{I - \sum_{j=1}^{j} NA_{ij}}$$

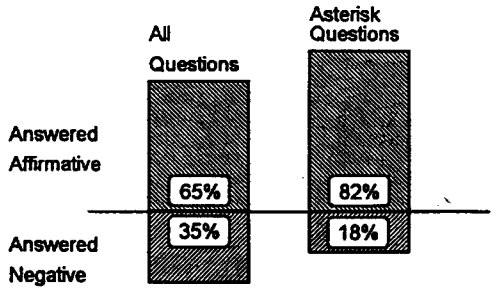


Figure 8-15 Adjusted Software Process Questionnaire Percentiles, Level 2

where the variables have the same meanings as in Equation 1. A spreadsheet can be used for this computation and to produce the pie graphs.

Aging of Metrics

Both the "*organizational profile*" and "*where we are*" metrics can be aged. This helps to eliminate the effects of older projects that may be operating using obsolete practices that prevent them from ever improving to Level 2 or 3.

A weighting factor (w_j) based on the age of the project is computed. The weighting factor should force the scores for projects older than a specified age to zero. This factor also should cause scores for older projects to be weighted less than those of more recent projects.

$$W_j = \frac{A_{max}^2 - \left(\frac{A_j}{12}\right)^2}{A_{max}^2}$$

If $w_j < 0$; set $w_j = 0$

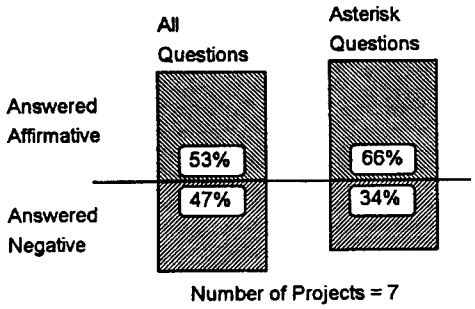
where A_{max} = Maximum age of project to have any score in years;
A_j = Age of project j in months;
w_j = Weighting factor for project j score.

For each category, the aged score (X) is then computed.

$$X = \sum_{j=1}^{J} \left(\frac{W_j \cdot P_j}{W} \right)$$

where
W = the sum of the weights

Figure 8-16 illustrates the effects of aging the fictitious projects with A_{max} = 5 years and projects 1 through 7 with ages of 9, 9, 16, 25, 36, 88, and 88 months, respectively. (Note that pseudo results for the questionnaires can be regenerated from Figure 8-15.) A spreadsheet may be used to perform the aged metrics computations and to generate the pie graphs.



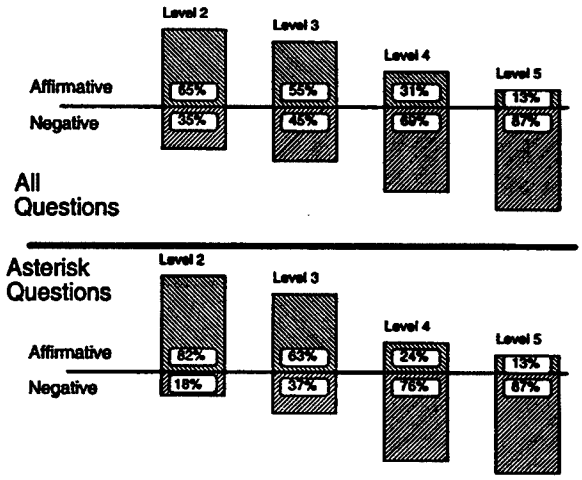
Requirements to Achieve Target Level
Achieve Previous Level
80% of All Questions Affirmative
90% of Asterisk Questions Affirmative

Figure 8-16 "Aged" and "Adjusted" Software Process Questionnaire Percentiles, Level 2

Figure 8-17 (below), which is also produced with the spreadsheet, shows the assessment results by asterisked and non-asterisked categories for each CMM level.

About the Authors

Steve Leadabrand and **Bill Bums** are actively working full-time in software engineering process improvement within the SEPG at Loral Vought Systems Corporation. Many of the company's products include real-time embedded software related to missile guidance and control.



Requirements to Achieve Target Level
Achieve Previous Level
80% of All Questions Affirmative
90% of Asterisk Questions Affirmative

Figure 8-17 "*Aged*" and "*Adjusted*" Software Process Questionnaire Percentiles On Seven Projects

S.J. (Steve) Leadabrand and W.E. (Bill) Bums
Loral Vought Systems Corporation
P.O. Box 650003, Mailstop EM-74
Dallas, TX 75265-0003
Voice: (214) 603-9628 (Leadabrand); (214) 603-9960 (Bums)
Fax: (214) 603-9629

CHAPTER 8
Addendum B

Software Complexity

Thomas McCabe

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 8
Addendum C

Metrrics: The Measure of Success

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 8
Addendum D

Making Metrics Work Miracles

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 8
Addendum E

Swords & Plowshares: The Rework Cycles of Defense & Commercial Software Development Projects

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

CHAPTER 9

Reuse

CHAPTER OVERVIEW

Better methods, advanced development tools, speedier hardware, more disciplined engineering, and improved management techniques cannot fulfill their promises, if we continuously build individual variations of the same basic software functions. For progress in software to occur, ***reuse must occur****. Reuse is a proven, effective way to increase productivity and reliability. Reuse occurs when software assets from prior programs are used again, or once built, the same assets are used more than once during the on-going development process. In most cases, reuse reduces the time and cost of developing new software — and the number of defects. Reuse is not limited to code alone. Opportunities for reuse abound with architectures, specifications, designs, test cases, documentation, and any other artifact you produce, or find in a reuse repository.*

Integrating software reuse into the software engineering process is a chance for dramatic improvement in the way software-intensive systems are developed and maintained throughout their life. [VISION92] *There are definite, quantifiable cost and time savings advantages to employing reuse. In this chapter you will learn that by implementing reuse at the beginning of the life cycle, the following benefits can be achieved:*

- *Improved quality and reliability,*
- *Quickly prototyped requirements through the use of existing components,*
- *Technical risks identified and managed early,*
- *Enhanced systems interoperability,*
- *Accelerated system development, and*
- *Reduced software acquisition costs.*

Blank page.

CHAPTER 9

Reuse

INCREASED QUALITY THROUGH REUSE

Obviously, the highest type of efficiency is that which can utilize existing material to the best advantage.
—Jawaharlal Nehru [NEHRU58]

The concept of reusing parts, components, and even major subsystems (common in the design and production of hardware systems) has been around a long time. Reuse saves time and resources and reduces the risk of building in defects. Effective reuse of knowledge, processes, and products from previous software developments has the potential for increasing DoD software productivity and quality an order of magnitude. In addition, the use of reusable software assets is a criterion for a development organization's progression to higher maturity levels. [CALDIERA91]

Software reuse is a process by which software assets (or components) are used in more than one application. Reuse can occur within a system (e.g., F-16A to F-16B), across similar systems (C-141 to C-17), or in widely different systems (AH-64 Apache to F-22). [CARDS92[2]] Software assets include source and object code, design documents, specifications, requirements, test cases, test code, test support data, users' manuals, programmer notes, algorithms, plans, and metrics — to name a few. ***The reuse of a software asset also implies the concurrent reuse of those items associated with it.***

Software must be designed to make maximum use of existing software and software products should be developed for subsequent reuse to the maximum extent possible. This means that you must identify and exploit software reuse opportunities, both Government and commercial, before beginning any new software development.

Therefore, you should base your reuse strategy on a determination of whether your software falls into one of the following categories.

- **A unique development**. A software asset is developed for a singular, unique purpose (i.e., no reuse).
- **Developed to be reusable**. A reusable software asset is developed to satisfy a particular need and can be reused for other than its original purpose and intent.
- **Reusing existing assets**. Existing assets are used as is, as COTS, GOTS, or NDI.
- Any combination of the above.

REUSE PROCESS

Four primary activities occur in the reuse process: domain engineering, architecture development, applications engineering, and reusable asset management. As illustrated in Figure 9-1, **domain engineering** identifies and develops a **product-line architecture** (occurring during the domain engineering phase) and develops and/or certifies reusable assets (e.g., requirements, design, code) within the domain architecture. **Application engineering** uses the assets from the domain engineering phase for developing new applications within the domain. Application engineering also identifies assets for reuse after they have been used and modified for new applications. These are then transferred into the domain engineering mode. Domain engineering can be categorized as ***"engineering for reuse"*** as it represents the supply side of reuse, where application engineering is the demand side. These two processes are linked by the **reusable asset management** process that serves as the middleman (or supplier) of reuse. This includes one or more repository, tools, and procedures required to manage reusable assets. [BLUE92] Figure 9-2 (below) illustrates how these activities are iterative within the reuse process.

NOTE: See Chapter 2, *DoD Software Acquisition Environment*, for a discussion on open systems, standards-based architecture. See Chapter 4, *Engineering Software-Intensive Systems*, for a detailed discussion of domain and applications engineering. See Chapter 14, *Managing Software Development*, for a discussion on architectural design.

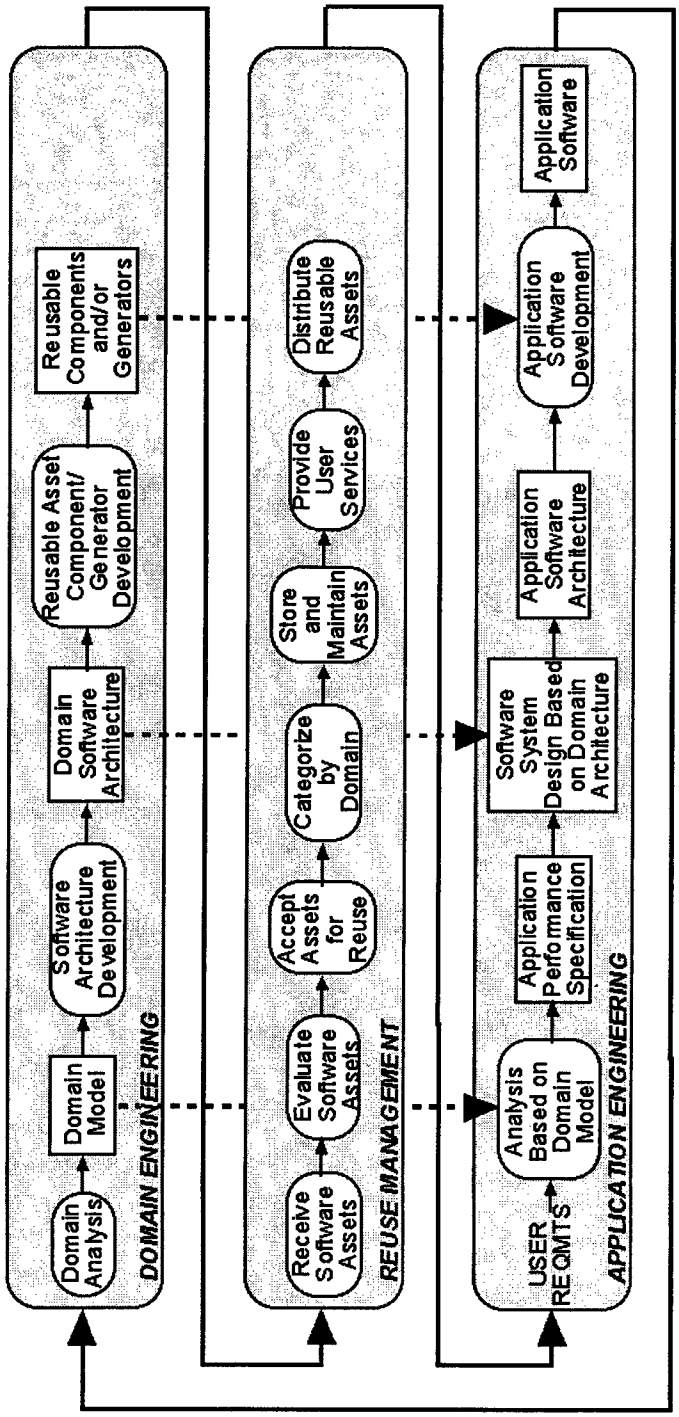


Figure 9-1 Domain-specific, Architecture-based Software Engineering to Maximize Reuse

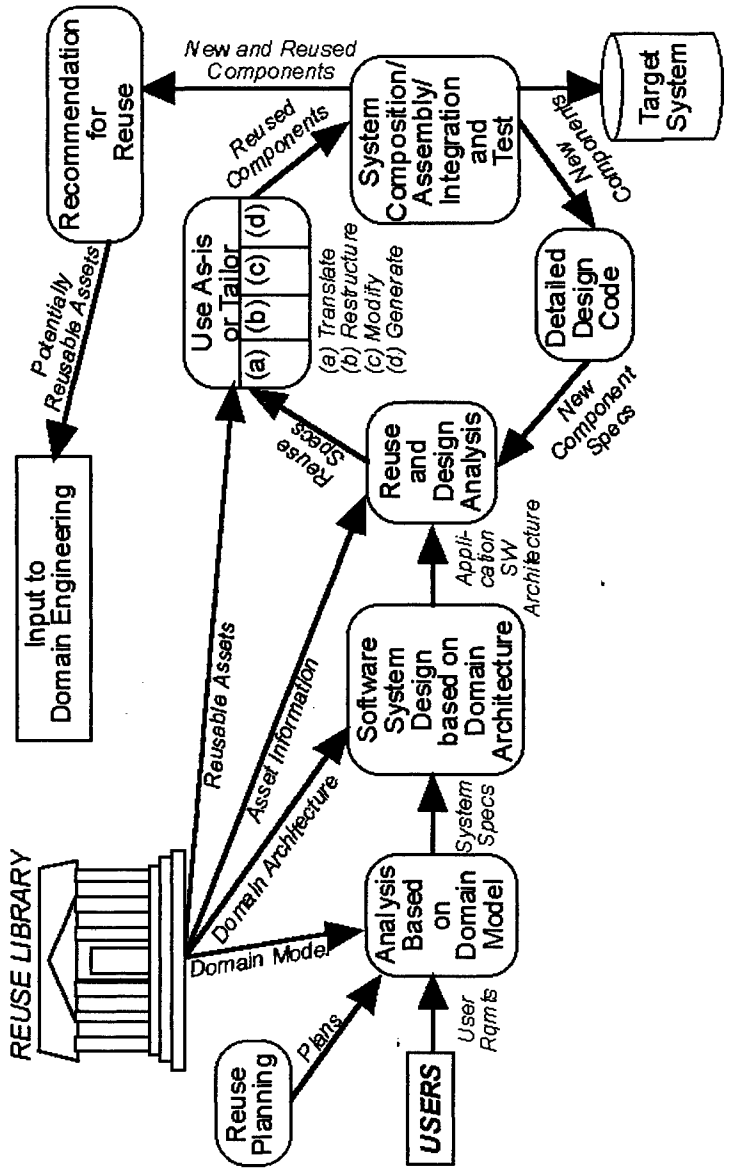


Figure 9-2 Reusable Asset Production through Iterative Domain and Application Engineering

Reuse Management Systems (Repositories)

The purpose of a **reuse management system** is to evaluate, accept, maintain, and make reusable assets available for use *(reuse)* on individual software developments. As the primary support tool for software reuse, the repository supplies reusable software assets, similar to a retail outlet store — thus it is a software reuse *facilitator*. The items for consumption can be based on vertical domains, horizontal domains, the types of components it contains, and the software language used. Merchandise can include software assets (e.g., code, specifications, designs, and documentation), software tools, or just a catalog of asset descriptions and sources. The retail store/repository analogy can be carried one step further.

- Store floor and shelf space = **capacity**,
- Boxed, wrapped merchandise and content descriptions = **packaging**,
- Store entrance and special sale notices = **access**,
- Walking the aisles and comparative shopping = **browsing** and **selection,**
- Check out counter and merchandise bagging = **extraction**, and
- Unit pricing and self-service shopping = **cost incentive**.

IMPLEMENTING REUSE

> *Most people have spent their lives reinventing the wheel, then refusing to concede that it's out of round.*
> [LENARD92]

Since software's inception, practitioners have been reusing algorithms, subroutines, and other chunks of code. They have also reverse-engineered legacy systems *[discussed in Chapter 11, Software Support]* for future reuse. This has been performed informally and usually in an *ad hoc* manner as circumstances allow. This type of reuse is referred to as *opportunistic*. The issue with reuse is not that it is not practiced. The problem has been that we have lacked an organized, systematic, conceptual framework and strategy for reuse.

DoD established high-level goals and objectives for software reuse with the DoD Software Reuse Initiative (SRI) Vision and Strategy. These goals were endorsed by Congress in 1994 with direction to form the **DoD Software Reuse Initiative Program**. The Air Force

has been supporting reuse efforts since 1991 and published the first version of their Software Reuse Implementation Plan in 1992. The plan has been used by various Air Force organizations to plan their reuse strategies. Since the initial plan was written, DoD and industry have significantly adjusted the focus of reuse strategies. It is, therefore, your responsibility to insert reuse technology into your software development and acquisition processes. This includes creating and using reusable assets, and working with the appropriate domain-specific reuse repository to exploit the benefits of their reusable assets.

The key to implementing architecture-based reuse is the selection of a **Reuse Project Officer (RPO)** who is an experienced software engineer that can understand abstract domain concepts and communicate with functional experts. The RPO's responsibility is to develop and exercise the software **Reuse Implementation Plan**. The plan should define the objectives, tasks, roles, and metrics to evaluate its progress. Reuse activities can be categorized as asset creation, asset management, or asset utilization. The reuse plan should organize reuse objectives around these activities. The plan must also address how to improve the reuse process and products through metrics and data collection to measure reuse success.

During the requirements phase, a thorough search should be made of all Ada repositories for candidate modules/packages for inclusion in your system. To fully reuse software, a one-to-one correspondence is needed between specifications, requirements, design, code, and test procedures. With this correspondence, not only can code be selected from a repository, but its associated documentation can also be reused. This correspondence is also helpful to the user. It defines what the software does and helps determine whether its design is compatible with its proposed use. The reuse capabilities of newly developed Ada code, in conjunction with the repositories, provide an opportunity for considerable savings in both time and money.

Product-Line Approach

A systematic reuse strategy defines a software production process that takes advantage of previous developments and focuses on enhancing a **product-line**, rather than on creating a new product. Borrowed from the hardware manufacturing industry, a product-line approach provides a better way to manage development costs. In the hardware factory example, fixed factory costs (infrastructure, e.g., facilities, tools, equipment, training, research) are separated from variable product costs (raw materials, labor, utilities), as illustrated on Figure 9-3. Increased capital investment in the factory yields increased productivity — resulting in reduced cost per product. In the software factory, the same benefits can be realized by investing in the establishment of product-lines. Fixed software factory costs (infrastructure, e.g., SEE, domain engineering and management, process engineering, architecture, training, metrics) are separated from variable product costs (COTS licenses, applications engineering, integration, test/certification, configuration management). An investment in upfront fixed costs yields increased productivity — resulting in reduced cost per software product.

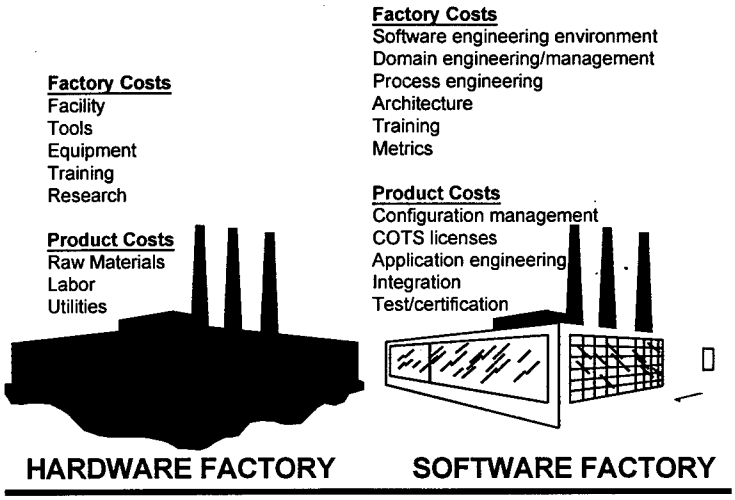


Figure 9-3 Factory Fixed and Variable Costs

A product-line is a suite of products that share common functional attributes (technology, design, parts, application processes, production methods). It is organized to realize *economies of scope*, but may consist of varying specific features and functionality required by

different groups of customers. Once a product-line is created, engineering efforts concentrate on defining the requirements for a particular version of the product and on how those requirements vary from the original product. The new product is built using the existing architecture, open systems standards, and object technology — all saving time during the design phase. Also, because the new version consists of pre-developed software artifacts, development time and testing are reduced. Any new artifacts, as well as any enhanced pre-developed ones, are placed in a reuse repository for future product versions. Through iterative reuse and development, the product-line constantly evolves into a higher quality, more advanced product.

One example of a product-line is that created by a telephone manufacturer. Their product-line may be organized into cordless hand-held, cellular/mobile, traditional, and multi-user/multi-lined networked telephones. Although their features and functionalities may differ, all telephone products share common technology, parts, and manufacturing processes. Examples of commercial software product-lines include: Hewlett-Packard software for laser-jet printers; versions of Lotus Development Corporation's 1-2-3 spreadsheet; Boeing Aircraft Company's commercial jet aircraft line; virtually all major automobile manufacturing; Intel Corporation's microprocessor line; and most commercial software packages. DoD also has a large number of potential software product-lines it acquires mostly through contractors. These systems (products) are categorized into domains and product-lines within those domains. Examples of DoD product-lines include avionics systems, radar systems, communication satellites, trainers and simulators, command and control systems, medical information systems, missile guidance systems, and personnel management and financial systems.

Product-Line Benefits

The transition to a DoD product-line architecture for the development and acquisition of software-intensive systems has the potential for significant benefits — shorter time to field, lower costs, greater interoperability, and improved quality. In fact, cost and schedule savings of as much as 65% have been reported. The architecture-based product-line approach institutionalizes the following software best practices:

- Open systems,
- Ada,
- Software development maturity,

- Business process improvement,
- Reuse,
- CASE tool use, and
- Metrics.

The benefits of a product-line approach include the following.

- Risk is reduced by reusing proven components and through development consistency;
- The amount of new development is reduced, while the number of products produced increases;
- The grouping of features can be maximized to increase the appeal of more customers;
- Time-to-market is decreased through reuse of technology, design, and components;
- Changes focus on lower-level software issues;
- Requirements growth and changes are addressed during the development process;
- Maintenance is easier and less costly;
- Rework is decreased through greater user/developer team focus;
- Recurring costs are converted to non-recurring costs;
- Cost of ownership is reduced;
- User/developer productivity is improved;
- *Economies of scale* are achievable through domain models, reusable components, and common SEE and mission environments;
- Stovepiped people problems are reduced:
  – Training is simplified (i.e., a lesson-learned is a lesson-shared),
  – "*Crunch-time*" staffing is decreased; and
- Smaller teams are required which reduces the lines of communications and resource expenditures.

An example of these benefits is the Navy STARS program. Early findings show productivity results for all programs are at $162/LOC. Subsequent development efforts within the domain have improved to $81/LOC. With the current build 85% of the requirements have been met and productivity improvements have reduced the cost to $65/LOC. Similar benefits have been documented for quality and cycle-time. *[NOTE: See the Air Force STARS SCAI discussion below.]*

Product-Line Paradigm Shift

For major DoD software-intensive systems with a common set of requirements, an architecture-based, product-line acquisition has great potential — but there must be a considerable paradigm shift for both the Government and its contractors. Reuse-based software development requires that both parties look beyond today's contract toward future acquisitions and mission needs. This new perspective demands a strategic vision wherein today's development serves as a baseline for reducing cost and schedule on future acquisitions. A strategic product-line plan must transcend acquisitions — rather than just focus on the current requirement. [CHRISTENSEN94] However, the software community alone cannot develop architecture-based product-lines. This paradigm shift will require upfront capital investment for the development and maintenance of the architecture, and for developing and certifying reusable components. Not only software developers, but comptroller, program evaluation and analysis, and acquisition organization executives must understand and commit to the creation of product-lines.

OPPORTUNITIES FOR REUSE

The opportunities for reuse are vast and not limited to code alone. Any and all software artifacts you produce or find in a reuse repository that meet your program-specific needs has the potential for cost-saving reuse implementation. These artifacts can include architectures, data, specifications, designs, test cases, modules, subsystem components, COTS, interfaces, or documentation, algorithms — to name a few.

Reuse for Embedded Weapon Systems

Embedded weapon systems often have such unique requirements that they are one of a kind developments. In these cases, software reuse is more *tactical* than strategic and often internal to the program. Embedded weapon systems have the following characteristics:

- Complex, real-time applications,
- Unprecedented requirements and technology,
- Highly integrated systems,
- Application-specific hardware and architecture (e.g., radar signal processors, flight control systems),
- Concurrent and integrated hardware and software subsystem development,

- Extensive development, integration, and combined hardware/ software test efforts,
- Long time spans from the development of one system to its successor,
- Integration of COTS, government-off-the-shelf (GOTS), and concurrently developed subsystems (e.g., armament and avionics systems).

Even with these characteristics, software reuse can offer benefits through the implementation of existing domain-specific software artifacts. Tactical reuse is *opportunistic* and can provide a way to minimize new development. Also, for embedded systems, functional reuse at the system and subsystem level may have higher pay back than a software-level approach. Typically they are required to meet strict real-time, safety-critical processing and interface requirements integrally related to the system's hardware. However, often there are many common modules, such as data retrieval or display routines, that can be used by all team members. Such vertical reuse can substantially reduce costs and shorten schedule. In fact, with the use of a mature Ada software engineering environment (SEE) you might achieve reuse of 20 to 30% within your present program.

The example of an aircraft flight control management system illustrates why embedded software reuse is sometimes limited. This subsystem translates pilot control inputs into aircraft control surface movements while factoring in aircraft attitude, speed, angle-of-attack, center-of-gravity, and control laws that prevent over-stressed conditions or loss of control by limiting the pilot's authority. It performs these functions through a management system that monitors numerous redundant aircraft-specific sensors and control features with finite dynamic ranges, sensitivities, noise reduction methods; and critical timing requirements. Adapting an existing flight control system for reuse in another aircraft involves significant effort in terms of understanding and modifying requirements, design, and testing to account for differences the new system's functional requirements, hardware, and architecture.

To compound the problem, major new developments or upgrades of one weapon system to another similar one are often decades apart. The time from the development of the **B-52** to the **B-1**, and from the B-1 to the **B-2**, is measured in decades. The same is true for the time span from the **F-15/F-16** to the **F-117**, and again to the **F-22**. During these times, the advancement of hardware, software, and weapon system-unique architecture technology, as well as greater user

expectations, work against the probability of successfully reusing software from a predecessor system. Aside from these obstacles, there are various approaches for achieving embedded system reuse.

- **Existing subsystem-level reuse**. When appropriate, existing subsystems are used as-is or modified to satisfy new or modified system requirements. This decision must be based on a series of engineering trade studies where each alternative's cost/benefits (including operational requirements) are identified and quantified, resulting in a recommended approach.
- **Develop for subsystem-level reuse**. Common subsystems are developed once and then reused on multiple platforms by meeting common interface requirements. The Standard Flight Data Recorder is an example of this approach which was designed to perform essentially the same function for all aircraft. [This illustrates why it is much more effective to reuse the entire subsystem (e.g., hardware, software, interfaces) than simply just the software.]
- **Intra-program software-level reuse**. Used on the F-22 program, this involves software-level reuse within an embedded weapon system development program and across the program team. Contractor teams use standard software engineering processes, as well as a common toolset. This includes identifying common requirements and developing common software components which are then available to all team members. This places the responsibility for reuse on the contractors, not on the Government.
- **Inter-program software-level reuse**. This type software-level reuse (addressed by the **1992 Air Force Software Reuse Implementation Plan**) involves reusing software components from one software system in another. The cost/benefits of this approach should be assessed through trade studies conducted by individuals knowledgeable about both systems. To make an educated assessment one must understand the existing system, its environment, and limitations, as well as the new system's requirements. With a clear view of both systems, a realistic evaluation of potential cost/benefits in reusing existing software artifacts can be made.
- **Architecture reuse**. This involves the reuse of software structures. For instance, structural modeling provides opportunities for software reuse across and within families of flight simulators. Structural models, comprised of multiple instances of a small number of structural elements, are characterized by multiple instances, or "*ensembles,*" of those elements. Complex interactions emerge from simple interaction patterns among these small number of elements. Structural modeling has been successfully employed

on the B-2 and C-17 training simulators, and has subsequently been transitioned to several other Air Force, Navy, and NASA programs.

- **SEE/toolset reuse**. This application domain offers the opportunity to exploit COTS and reuse. For example, the F-22 program is reusing a common system/SEE (S/SEE) across all major prime and subsystem software contractors. Advanced development tools generate code for multiple, standard processing platforms and consistent application interfaces which help simplify software-level reuse.

Specification Reuse

An efficient method for improving the specification process is through reuse. **Specification reuse** can completely eliminate the effort involved in designing, coding, and testing the implementation of a specification. Specification-level reuse includes, for example, specification models, data flow diagrams, state-transition diagrams, and structured English process specifications. [YOURDON92] You should make sure your developer is aware of, and browses, the DoD and Air Force reuse repositories for specifications that might be used as a starting point for your program.

Architecture Reuse

Architecture reusability is based on the premise that there is a commonality among the architectures of similar type software systems. For instance, a commonality exists for all C2 systems and another core commonality exits for logistics systems. These commonalties are represented in **generic software architectures** that guide the design of systems within the domain. Commonalties must be factored out and dealt with in the broadest context possible. As the software architecture implements solutions to the **problem domain**, ***it becomes a model for constructing applications and mapping requirements from the domain model to reusable components.*** A generic architecture provides a high-level generic design for a family of related applications, as well as a set of components that can be reused for any instance of that application. A generic design also eliminates the need to develop a high-level design for each application within the domain. Domain developers use these generic representations as specifications for reusable components. [PAYTON92]

Design Reuse

When one thinks of reuse, it is usually in the context of *reusable code*. The problem with concentrating only on code is that code is produced after the most difficult phase of software development has been performed—requirements analysis and design. ***Coding should only consume about 10% to 15% of total software development time and effort***, so increasing coding productivity and quality (through reuse or HOLs) can only achieve limited productivity increases. Good designers usually use design analogies from past problem solutions, and often have a repertoire of past successful design solutions upon which to draw. This is why the ***skills, experience, and resources of your developer with programs of similar size, complexity, and problem domain are crucial to this process***. Hence, the first version of the model that goes through the simulation process is usually not a raw, untried, new model, but a viable candidate borrowed from a past solution. **Reuse of designs** at the developer level is a very common and efficient practice. [GLASS92]

NOTE: Identifying the skills, experience, and resources needed from your developer can be a difficult value judgment. The Skills Matrix *[discussed in Chapter 13, Contracting for Success]* **is a useful method for assessing whether contractor personnel qualifications will give you the depth of experience and breadth of domain knowledge your development requires. A developer with extensive experience in your domain is essential to success!**

While code reuse typically occurs only at lower-level system design hierarchies, design reuse often results in whole branches of the hierarchy being reused. The darkened circles on Figure 9-4 represent modules with either design or code reuse.

Code Reuse

When developing code for reuse, special attention must be given to the parameters and structure of the software units to be reused to isolate specific hardware and system dependencies. Ada packages help in this area, but the code must be designed carefully so that the reusable unit is as generically applicable as possible. As the units of reusable code increase and become more generic, you will need more automated tools to keep track of reusable pieces. In some cases, the

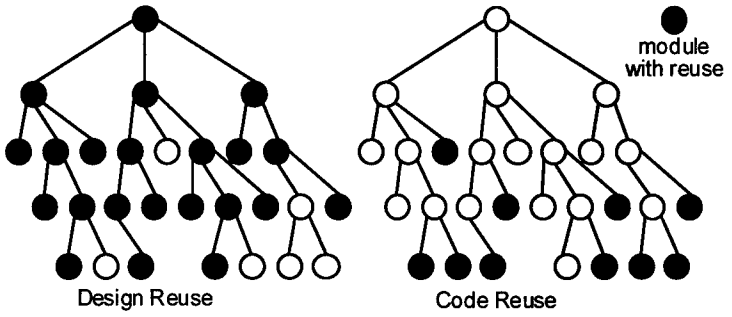


Figure 9-4 Design Reuse Compared to Code Reuse [YOURDON92]

automated tools to build reusable systems are as complicated (or more so) than the systems they produce.

There are three general types of code reuse: (1) **cut-and-paste** of source code; (2) using the mechanism contained in most HOLs for **copying** or including source code text from a library; and (3) **binary links** where already compiled external subroutines, procedures, or functions are contained in a library. Binary links can be incorporated into the application by invoking them through a *linking* mechanism after the main application has been compiled. The advantage of this third form of code reuse is that only one physical copy of the reused component is needed, regardless of the number of times it is invoked. Ada features and the structure of Ada code greatly increase the potential for code reuse. [YOURDON92]

NOTE: Binary links to routines in a library are the only option that will ultimately reduce the software maintenance burden by minimizing the overall volume of software maintained by DoD. The other two options may reduce development costs, but multiple copies of reusable code require the same maintenance costs as new code because the overall volume of the software has increased. A reuse goal should be to take advantage of maintaining a single source component and using multiple executions.

Ada Reuse

In addition to Ada's ability to enable software engineering goals through design features that support software engineering principles, Ada design features also facilitate the economic benefits of reuse. *[See Chapter 5, Ada: The Enabling Technology, for a discussion on Ada features.]* By exploiting Ada's reuse features, significant gains in software productivity and quality can be achieved.

The **information hiding** found in well-designed Ada packages has a positive impact on reuse by containing few input parameters. The less you need to know about the environment external to the package component, the more likely the package can be reused. A common example is the abstraction, or hiding, of device-dependent logic by other portions of the Ada program. These other portions can be easily reused by different devices. Also, Ada software can be reused at other than the component level. That is, entire Ada programs or collections of Ada programs can be reused. Reuse of entire programs simplifies many reuse issues. Configuration control is streamlined because all users receive the same software version. In addition, production planning can be simplified through a periodic release cycle.

Data Reuse

With greater reliance on **CASE technology**, a CASE repository provides an excellent opportunity for data reuse. Not only are data declarations reusable, but so are all types of data definitions (such as data flow diagrams, entity-relationship diagrams, physical database designs, and structure charts). Code and parameter definition reuse are the most common form of reuse facilitated by a CASE repository.

COST/BENEFITS OF REUSE

As you learned in Chapter 8, *Measurement and Metrics*, a significant variable affecting the cost of software is the number of source lines-of-code (SLOC) developed. The impact of this factor is straightforward — the greater the size of the system, the more expensive it is to build. Reducing the number of SLOC is an area ripe for process improvement. The easiest way to reduce the number of SLOC developed is through ***reuse.*** With reuse, the cost not only goes down, but quality and productivity go up. Gains in productivity, however, are never on a one-to-one ratio with reused resources. An

organization achieving 80% reusability is not necessarily four times more productive than one achieving 20% reusability. Productivity savings will be somewhat offset by the dollar investment in the product-line infrastructure which reuse requires.

Cost of Reuse

A discussion on the **cost of reuse** naturally migrates to the **cost-of-quality**. More testing and quality assurance is required because the consequences of a latent defect become increasingly more serious with reusable assets. On the other hand, heavily reused software assets have, by definition, higher quality than custom assets because defects get shaken out quicker and more thoroughly. ***Remember, quality does not cost — it pays!*** If you build a system using a large percent of reusable assets, you will be building a system that merits the *"People's Choice Award!"* [YOURDON92] Reuse cost considerations include:

- The investment in creating reusable assets must be amortized over the number of new systems or programs which can make use of those assets. Obviously, the more assets that get reused, the less burdensome is the investment.
- Reuse requires that higher levels of testing and quality assurance than normally performed for custom software components. These higher levels are warranted by higher levels of usage. (On an average, 2 to 4 times more testing is required for reusable components.)

The assumption that building for reuse is expensive often comes from organizations with immature software development capabilities. Most methods for increasing reusability (i.e., modularity, high cohesion, low coupling, use of standards) are good software engineering practices routinely used by mature organizations. This is not to imply there are no additional costs associated with reuse. Although there is increased support through the reuse programs and repositories *[discussed below],* the cost to implement and maintain a database and retrieval system for internal reuse on a major software development can be substantial. Also, domain engineering, a relatively new discipline, requires training and manpower. Overly enthusiastic estimates of cost savings for reuse-based development and underestimating reuse implementation costs are common. A list of reuse costs should include:

- Domain analysis and modeling,
- Domain architecture development,
- Inspection and quality assurance of reusable components,
- Increased documentation to facilitate reuse,
- Maintenance and enhancement of reusable assets, and
- Training of personnel in design and coding for reuse. [CHRISTENSEN94]

NOTE: See "*Reuse at Hewlett Packard*" below for a discussion on the cost of reuse.

There is one form of reuse that is almost free — reuse within your program. Experience shows that in a well-managed software development, using Ada and an associated robust software engineering environment [e.g., Rational/Apex Environment™ *(discussed in Chapter 10, Software Tools)*], reuse of up to 35% can be achieved. Be sure that your software development team has the vision and tools to obtain maximum reuse leverage from the software modules they themselves develop.

Benefits of Reuse

The benefits of reuse can be substantial. For a specific acquisition, the dollar savings can be calculated by estimating the cost for development without reuse and subtracting the cost for development by reusing existing software assets. While this calculation may not be that precise, it provides a reasonably accurate estimate that indicates an order of magnitude potential savings. Because reusable software quality is generally known and substantially higher than newly developed software, a degree of confidence can be achieved through familiarity with a known entity. Reuse benefits includes:

- Increased productivity,
- Shorter development schedule,
- Reduced costs over time,
- Increased quality and reliability,
- Earlier requirements verification,
- Lower risk through more accurate size estimates,
- Higher ability to leverage expertise in individual domains, and
- Shorter time to field.

CHAPTER 9 Reuse

The greatest savings in reuse are achieved through systematic, strategic reuse-based development. In 1992 the **Advanced Research Projects Agency (ARPA),** projected software costs and savings to the year 2012. As illustrated on Figure 9-5, the savings anticipated in reuse-based software development will equals the combined savings of process improvement and software development tools.

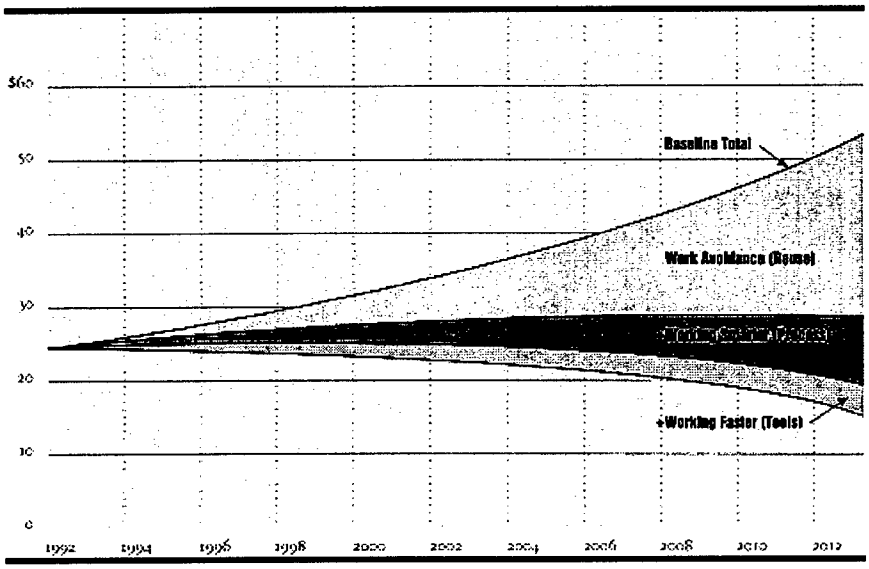


Figure 9-5 Annual Projected DoD Software Cost (dollars in billions) [CHRISTENSEN94]

The **Software Engineering Laboratory (SEL)** (run jointly by NASA, Computer Sciences Corporation, and the University of Maryland) tracks aerospace software programs and maintains a database of program statistics. As illustrated in Figure 9-6 (below), the study of 887 **Fortran programs** showed that 98% of reused modules were defect-free compared to only 44% of new modules. In addition, this study (and others conducted by the SEL) indicates that code reuse requires only 20% of the cost required for new code. [FISHER91] *[Subsequent studies have shown even greater benefits than the above when Ada is used compared to Fortran programs.]*

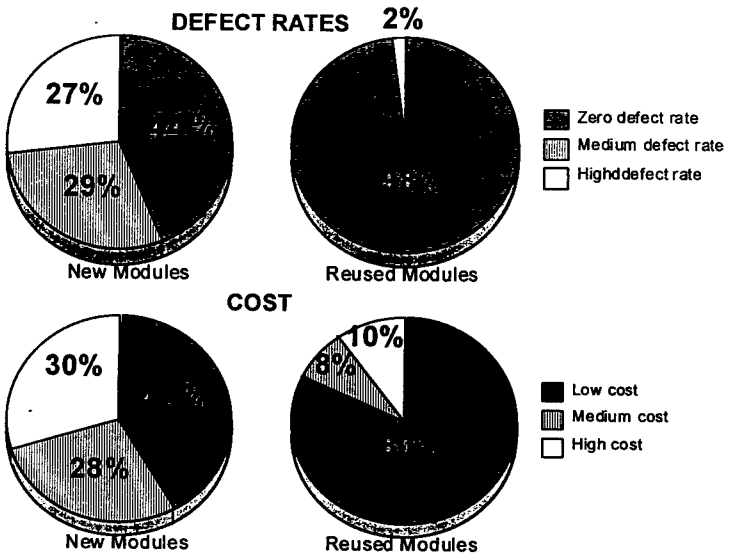


Figure 9-6 SEL Comparisons of Defect Rates and Development Cost of New and Reused Modules

Reuse at the Standard Systems Center (SSC)

As discussed in Chapter 7, *Software Development Maturity*, a study report was published in 1993 about the software engineering process at the Standard Systems Center, Maxwell AFB-Gunter Annex, Alabama. One of the study's objectives was to determine the economic benefits of reuse from a library of standard objects inherent in an I-CASE environment. The researchers determined that the maximum benefit would be if a program went from a Level 1 to a Level 3 with 70% reuse. While this could not be observed with actual program data, the data from the same SSC program were used to predict the benefits of reuse, as illustrated on Table 9-1. They found when process improvement is combined with effective reuse of Ada, even higher levels of savings/productivity and improved reliability are possible.

The report concluded that significant reductions in cycle time (where cycle time is defined as the time to take a request for a new capability and field that capability) can be achieved by effective exploitation of reusable libraries of proven modules (objects) in modern Ada, I-CASE, and other object-oriented environments. Reduced cycle time benefits

	NORMAL	WITH REUSE	DIFFERENCE BENEFIT	PERCENT IMPROVEMENT	BENEFIT RATIO
Size (SLOC)	162,093	48,628	113,465	-70%	3.34
Schedule (Months)	24.50	8.6	15.9	-65%	2.85
Effort (PM)	1494	52	1,442	-97%	28.98
Cost ($K)	$5,716	$197	$5,519	-97%	28.98
Staff (People)	100	9	91	-91%	11.11
MTTD/Days	0.43	2.37	1.94	451%	5.51

Table 9-1 Impact of Reuse and Moving from Level 1 to Level 3 (113,465 SLOC from reuse)

can also be realized in effort/cost and reliability (fewer defects and longer MTTD). Table 9-2 illustrates the benefits of reuse at 70%) for an SSC program at a Level 3.

	ACTUAL	WITH REUSE	DIFFERENCE BENEFIT	PERCENT IMPROVEMENT	BENEFIT RATIO
Size (SLOC)	162,093	48,628	113,465	-70%	3.34
Schedule (Months)	14.32	8.6	5.72	-40%	1.67
Effort (PM)	263	52	212	-80%	5.11
Cost ($K)	$1,008	$197	$811	-80%	5.11
Staff (People)	31	9	22	-71%	3.44
MTTD/Days	1.38	2.37	0.99	72%	1.72

Table 9-2 Impact of Reuse at SEI Level 3 (113,465 SLOC from reuse)

Reuse on the F-22 Program

The F-22 production contract requires the implementation of a software development reuse program, of which the F-22 SPO is an active participant. The three prime contractors (Lockheed Aeronautical Systems Company, Lockheed Fort Worth Company, and Boeing Military Aircraft) and multiple subcontractors are networked through a common system/software engineering environment (S/SEE). Electronic conferences among team members

distribute reuse information (e.g. reuse item description, name of original developer(s), currently known reusers, and security requirements).

F-22 software reuse is not limited to operational flight programs (OFPs), but includes compilers and support tools such as debuggers which are combined into extensions hosted on the S/SEE. While engineering simulations are usually developed *ad hoc*, many are being developed according to a structured method and documented for future reuse. Specific requirements are also included in some simulations (and OFPs) to allow for reuse. Reuse agreements are documented between the developer and any known reusers on items such as budget, schedules, risks, and maintenance approach. Software reuse is being implemented on the F-22's weapon systems, air vehicle, support systems, and training systems, wherever practical, feasible, and cost-effective. [CHRISTENSEN94]

Reuse on the F-16 Upgrade Program

Contracts for the **F-16 Modular Mission Computer** and F-16 **Mid-Life Upgrade** programs require software reuse through the addition and deletion of functional components. This means software artifacts (design, documentation, and code) must be designed and developed for reuse. Future systems based on this design will benefit from substantial cost and schedule reductions through systematic reuse-based software development. [CHRISTENSEN94]

Reuse at Hewlett-Packard (HP)

Hewlett-Packard (HP) has collected metrics from many reuse programs which document improved quality, increased productivity, shortened time-to-market, and enhanced economics resulting from reuse. Because work products were reused many times, accumulated defect fixes result in higher quality products. Productivity has increased because reused products are previously created, tested, and documented. Reuse has reduced time-to-market when effectively placed on the development program's critical path. Personnel expertise has been leveraged because reuse allows experienced software engineers (reuse "*producers*") to concentrate on creating work products which less experienced personnel (reuse "*consumers*") then reused. Table 9-3 illustrates the results of reuse at two HP software development divisions.

	HP MANUFACTORING DIVISION	HP TECHNICAL GRAPHICS DIVISION
Quality	51% defect reduction	24% defect reduction
Productivity	57% increase	40% increase
Time-to-market	Data not available	42% reduction

Table 9-3 Relative Cost to Produce and Reuse [LIM94]

HP realized that software reuse is not free. Reuse costs include creating or purchasing reuse products, libraries, tools, and implementing reuse-related processes. Table 9-4 summarizes the costs of reuse on the **FAA's Advanced Automation System (AAS)** and two other HP programs. They found the relative cost of creating an AAS reusable component was about twice as much as creating a nonreusable one; and the costs to integrate reused components into new products was about 10% of the cost to create it. On the MIS system, the relative cost of producing a reusable component ranged from 120 to 480% the cost of creating a nonreusable one, of which integration costs ranged from 10 to 63% of the total cost. On the graphics system, the cost of creating a reusable component was 111% the cost creating a nonreusable one, of which integration costs were 19%.

	AIR TRAFFIC CONTROL SYSTEM	MENU AND FORMS MIS	GRAPHICS FIRMWARE
Relative cost to create reusable component	200%	120 to 480%	111%
Relative cost to integrate reusable component	10 to 20%	10 to 63%	19%

Table 9-4 Quality, Productivity, and Time-to-Market Profiles [LIM94]

Given the cost of reuse, it is essential to track the economic return on investment (ROI) an organization receives for its reuse efforts. To determine their ROI, HP developed a relative cost model that defines development with reuse as a proportion of a baselined program. The model's net-present-value method arrives at a cost/benefit value by subtracting the *producer* investment cost of making reusable work products from the saved net *consumer* development costs. Thus, the net-present-value method takes the estimated value of reuse benefits and subtracts them from associated investment costs, taking into account the time value of money. The model covers the entire life cycle (including maintenance), accounts for risk, and recognizes

potential increased profit from shortened time-to-market. Because shortened time-to-market is difficult to assess, HP uses a conservative overall economic benefit estimate. Table 9-5 illustrates the net-present-value economic analysis for the two HP software divisions analyzed.

	HP MANUFACTORING DIVISION	HP TECHNICAL GRAPHICS DIVISION
Time horizon	1983 - 1992 (10 years)	1987 - 1994 (8 years) 1994 data estimated
Start-up resources required	26 engineering months (start-up costs for 6 products) $0.3M	99 engineering months (~3 engineers for 3 years) $0.3M
On-going resources required	54 engineering months (~½ engineer for 9 years) ~$0.3M	99 engineering months (~1 to 3 engineers for 5 years) ~$0.7M
Gross cost	80 engineering months ($1.0M)	206 engineering months ($2.6M)
Gross savings	328 engineering months ($4.1M)	466 engineering months ($5.6M)
ROI (savings/cost)	410:1	216:1
Net-present-value	125 engineering months ($1.6M)	75 engineering months ($0.9M)
Break-even year (recoup start-up costs)	2nd year	6th year

Table 9-5 HP Reuse Program Economic Profiles [LIM94]

REUSE PROGRAMS

Consult the **Air Force Computer Systems Authorization Directory (CSAD)** for information on reusable assets in your domain. There are several reuse programs within DoD and the Air Force providing a network across all domains to assist in implementing reuse within your program. Major programs focusing on the institutionalization of software reuse include the following.

NOTE: See Volume 2, Appendices A and B for information on how to contact the following reuse programs.

Software Technology for Adaptable, Reliable Systems (STARS)

The **STARS** program, sponsored by ARPA, is contracted through the Electronic Systems Center (ESC), Hanscom AFB, Massachusetts. STARS objectives are to demonstrate the benefits of reuse, provide transition support to reduce the risk of incorporating reuse in the systems engineering process, and to ensure basic capabilities (process and technologies) are available for support. The STARS strategy is to change the way DoD develops software through ***megaprogramming.*** Instead of developing code line-by-line, megaprogramming methods and tools support the capture and reuse of functional blocks of existing code by integrating them into new applications. Megaprogramming involves defining a family of applications that share common traits and characteristics (a product-line), creating a product-line set of models, and building products based on the models while accumulating/using a set of domain-specific assets, as illustrated on Figure 9-7. Megaprogramming contains four elements:

- Process-driven development,
- Domain-specific reuse,

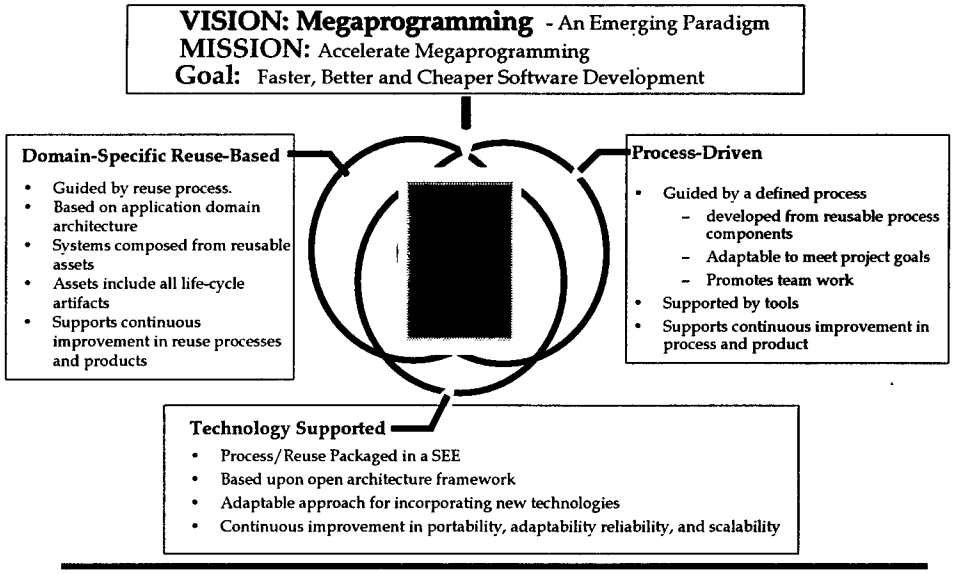


Figure 9-7 STARS Megaprogramming

- Technology support, and
- Collaborative development.

STARS process technology and transfer activities are organized to assist process insertion, usage, and improvement to promote process-focused development of major software-intensive systems. STARS process-based development occurs when:

- Organizational processes exist which are adapted and tailored to meet program and product goals;
- Software development is guided by a defined process;
- Environments and tools are integrated to support the defined process;
- The defined process promotes collaboration and teamwork by making activities, roles, and dependencies clear; and
- Discipline and automation support continuous improvement of the defined process through measurement and feedback.

The STARS conceptual framework for reuse supports the evolution of software engineering from *ad hoc* solutions to *"engineering discipline."* The process-based development model, illustrated on Figure 9-8, addresses the interactions between major participants in process development. These include process definers, process users, process improvers, and those responsible for process technology institutionalization. The main components of the model (i.e., the main STARS activities) are a process asset library, process definition technology, process enactment (use) support, process measurement technology, and process evolution concepts.

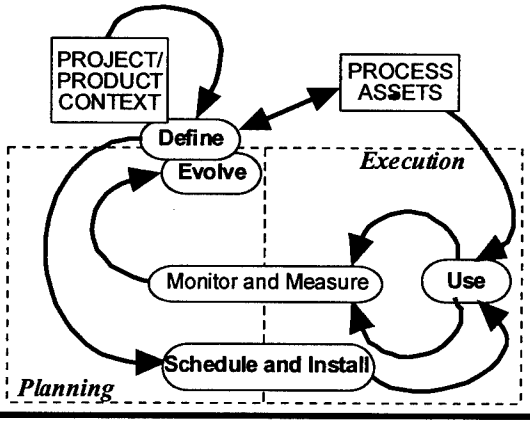


Figure 9-8 Process-based Development Conceptual Model

CHAPTER 9 Reuse

STARS has developed the **Reuse Strategy Model: Planning Aid for Reuse-based Projects**. This model provides an assessment of your developer's current reuse practices and identifies a set of possible goals that can be translated into realistic resource and schedule terms for your program. By conceptualizing measures to gauge progress against program goals. [RSM93]

The process-based technology sponsored by STARS is founded on SEI process concepts. *[SEI is the lead in joint STARS/SEI Process ASSET Library activities.]* STARS promotes automated tools for process-based technology (tools that support the emerging role of *"process engineering"*) by evolving commercially-supported software engineering environments. Ada systems development is the primary target for STARS technology, therefore, STARS sponsors formalizations of **Ada process models** and reuse processes that take best advantage of Ada's features.

NOTE: See Volume 2, Appendix L for descriptions of STARS products and services.

STARS Space Command and Control Architecture Infrastructure (SCAI)

The STARS **Space Command and Control Architectural Infrastructure (SCAI)** program (also known as the Air Force STARS Demonstration Project) the creation of a product-line at the Space and Warning Systems Center (SWSC), Peterson AFB, Colorado. Two command and control systems (one million LOC) were built in support of Cheyenne Mountain missions. A team of only 43 people (16 programmers) developed the systems in two years, achieving extremely low defect rates and reuse of better than 50%. The architecture-based product-line approach they implemented included:

- Megaprogramming concepts,
- Process technology,
- Two life cycle model,
- Cleanroom engineering,
- Architectural infrastructure,
- Object-oriented technology,
- Domain experience,
- Working applications,
- Enhanced technology, and
- Real program experience.

Prior to implementing the product-line approach, software development at the SWSC was performed in a stovepipe fashion where four systems were developed and maintained independent of each other. Cross-fertilization was unknown and the four teams worked in parallel performing the same tasks, maintaining the same spares, conducting the same training — all with lowered effectiveness. As illustrated on Figure 9-9, parallel efforts evolved into 34 separate operating systems, 27 languages, 12 millions lines-of-code, multiple proprietary hardware and software components, and complex SEEs.

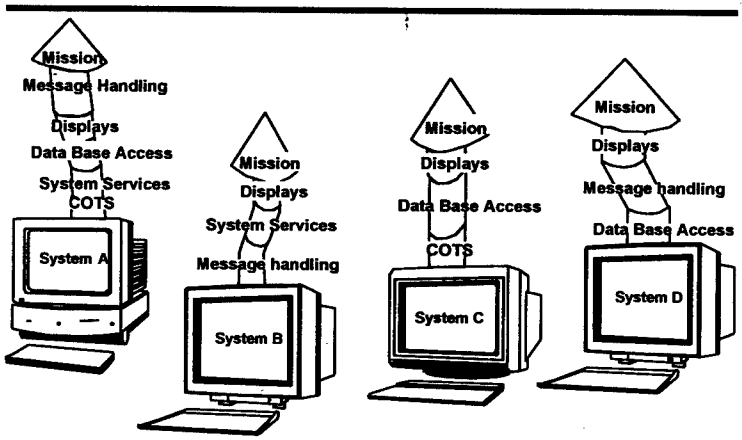


Figure 9-9 SWSC Stovepiped Software Systems

Domain engineering was used to alleviate their stovepipe dilemma by developing an architecture-based product-line. They defined three software domains (air, space, and missile) based on a C2 architectural infrastructure that identified a common set of services (system, message handling, data management, and user interface), as illustrated in Figure 9-10.

Another benefit the product-line approach achieved was the formation of a product-line organization. The former stovepiped organization was reduced from 200 people to a matrixed one consisting of 132 people which mirrored the system architecture, as illustrated on Figure 9-11. Personnel expertise was assigned and organized by architecture functional areas:

- Mission tasks were assigned to space mission experts,
- Applications tasks were assigned to astronautical engineers,
- Services tasks were assigned to system engineers, and

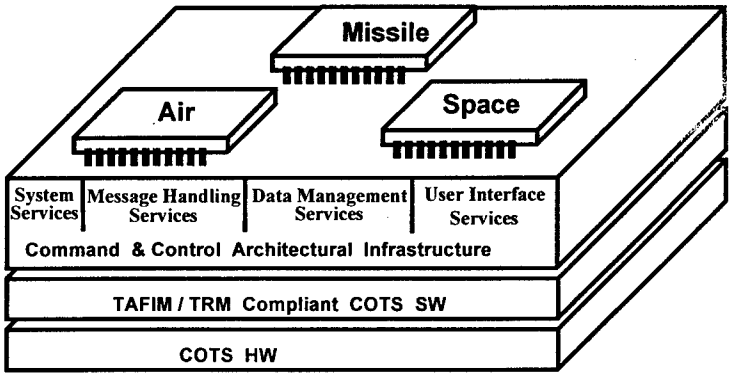


Figure 9-10 Architectural Infrastructure is the Product-Line Approach Foundation

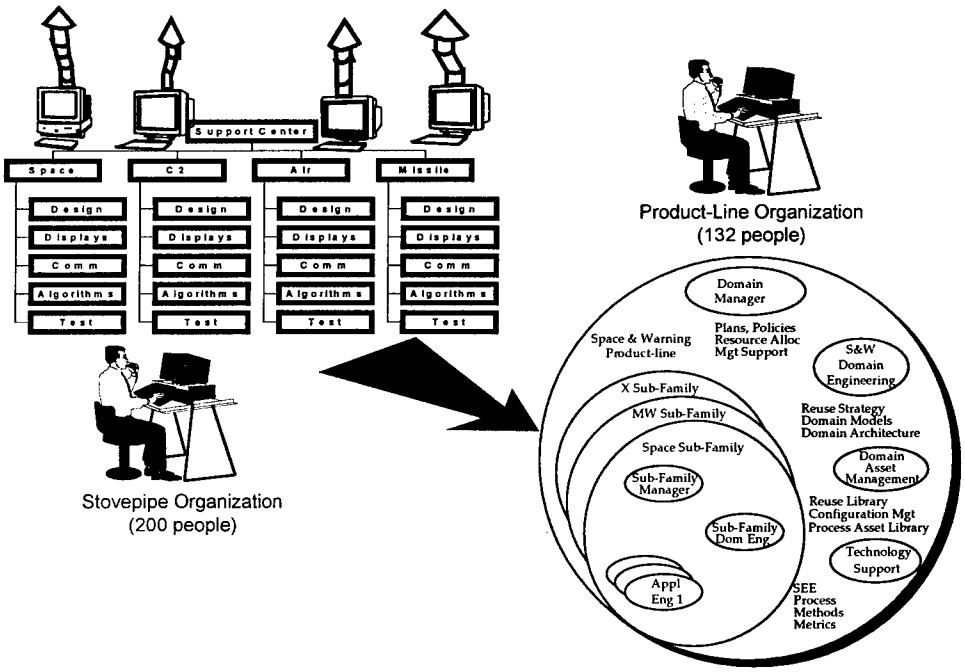


Figure 9-11 SWSC Product-Line Saved People Resources

- Application architectural modeling was assigned to system architects.

As illustrated on Figure 9-12, the SCAI **architecture-based product-line** contributed to increased reuse on the **Mobile Space Project**. On Build 2 of 3, hand coding was reduced to 36.5%, code generator reuse was 50.5% and architecture infrastructure reuse was 10%. Compared to traditional software developments, the resultant benefits of this approach were impressive, as illustrated on Table 9-6.

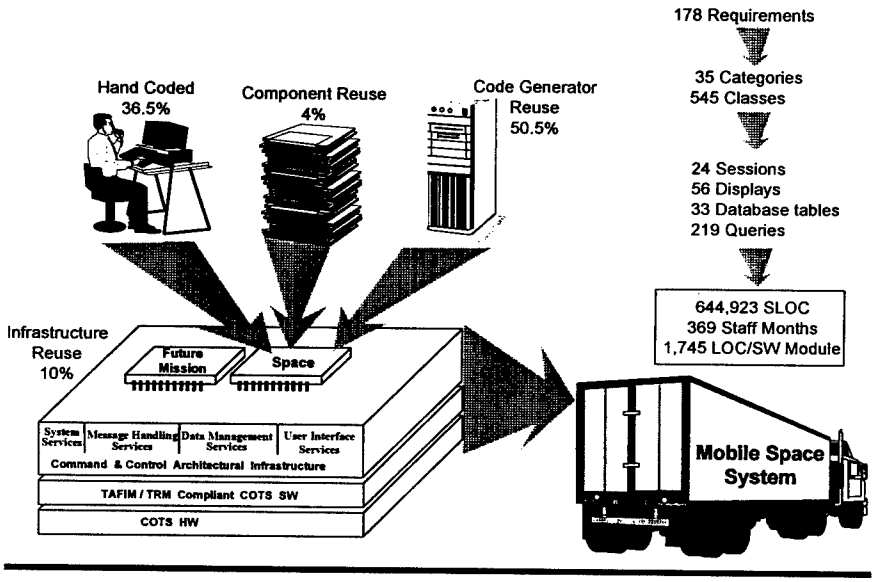


Figure 9-12 Reuse on SCAI Mobile Space System Build 2

	TRADITIONAL	MOBILE SPACE SYSTEM
Schedule	42 months	13 months
Quality	2 defects/KLOC	2 defects/10 KLOC
Cost	$130/LOC	$47/LOC

Table 9-6 Traditional versus Mobile Space System Product-Line Approach

The SCAI AF/STARS project is proof that a standards-based architecture is the foundation of the product-line approach and the missing link in the reuse equation, as illustrated on Figure 9-13.

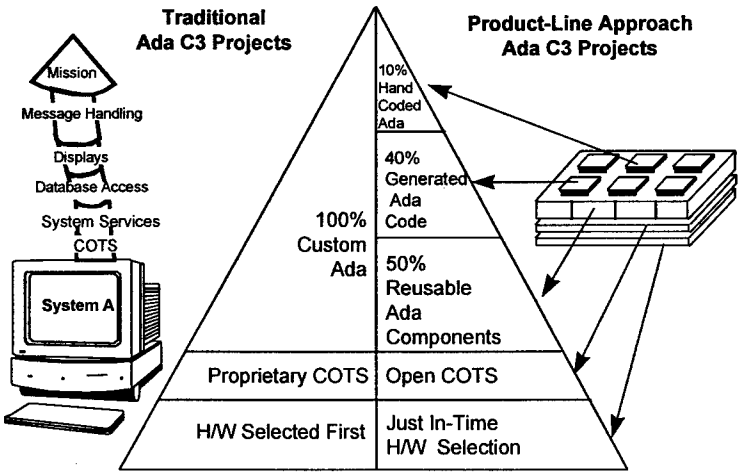


Figure 9-13 Architecture Is the Key to Reuse Success

NOTE: Contact DISA *[see Volume 2, Appendix A and B for information on how to contact them]* **about the Global Command and Control System (GCCS) Common Operating Environment (COE) and Global Combat Support System (GCSS) models.**

Lessons-learned on the SCAI program indicated that there must be a significant paradigm shift for both the Government and its contractors for product-line reuse to succeed. As stated above, both parties must look beyond today's contract toward future acquisitions and mission needs if we are to reap the cost and quality benefits of reuse. The obstacles the Air Force encountered to the institutionalization of this new paradigm are summarized on Figure 9-14 (below). Money is appropriated for stovepipe development, not infrastructure. We would rather pay less now, not realizing it is going to cost us much more later. They realized this is mainly due to the near-sighted nature of defense acquisition. A much bigger issue than one program can tackle, this must be dealt with at higher levels in the decision-making process. The vision that an increased capital investment in infrastructure and product-line technology will reap quantifiable savings in the long-term has still to be understood and embraced. But perhaps this understanding and the beginnings of the paradigm shift will trickle up from grassroots, real-life examples such as the SCAI program.

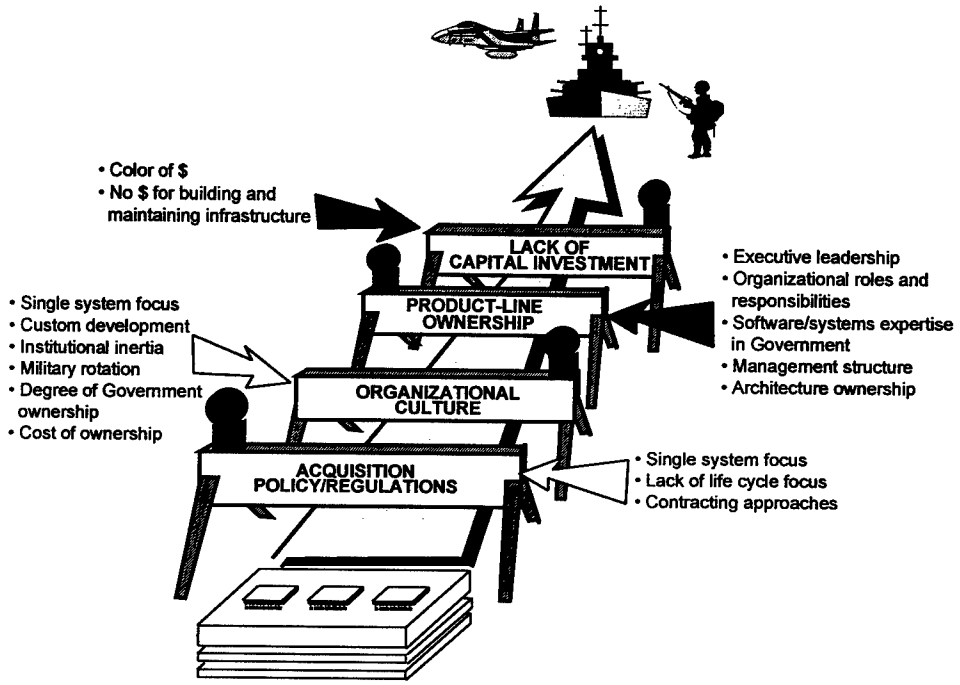


Figure 9-14 Obstacles to the Product-Line Paradigm Shift

NOTE: See Chapter 2, *DoD Software Acquisition Environment*, for a discussion on open systems architecture and the TAFIM. See Chapter 4, *Engineering Software-Intensive Systems*, for a discussion on SCAI domain engineering. See Chapter 16, *The Challenge*, Addendum "*Reflections on Success*" for a more complete description of the SCAI success story.

Asset Source for Software Engineering Technology (ASSET)

Asset Source for Software Engineering Technology (ASSET) offers products and services in digital support, electronic commerce, and software engineering with an emphasis on re-engineering and reuse. ASSET, established by the **Advanced Research Projects Agency (ARPA)**, as a subtask under the STARS program is transitioning to a private enterprise as a division of SAIC. ASSET offers the following support services to the software engineering community.

- **Library of reusable assets**. A universally accessible library of reusable software assets and digital products is available through the **Worldwide Software Resource Directory (WSRD)** which contains over 700 assets servicing over 1,500 users throughout the world. ASSET has established library interoperation with four libraries allowing our users Internet access to remote libraries. Through the WSRD, users can search, browse, and download assets cataloged in over 30 domains.
- **On-line reuse information**. On-line information of reusable technology, publications, and conferences is accessible through ASSET's World-Wide Web pages (see Volume 2, Appendix B for ASSET's Web address) which describe ASSET products and services, as well as information related to software reuse. ASSET's catalog with titles and abstracts is also accessible through the Web.
- **Reuse brokerage services**. Reusable software asset brokerage services and digital products, available through ASSET's electronic commerce capability, can be ordered as well as delivered electronically. ASSET acceps credit card numbers either over the Internet or via an "*800*" number. Orders are shipped electronically or via diskettes, tapes, or hardcopy.
- **Custom digital libraries**. Custom digital libraries are being tailored to special customer needs, either at the customer's site or at ASSET's site in Morgantown, West Virginia which is accessible through the Internet, the Web, or modem. The customer provides the assets and an access list which ASSET catalogs and makes available electronically while controlling library access. Monthly usage reports are provided to the customer.

Comprehensive Approach to Reusable Defense Software (CARDS)

CARDS has been assisting the Air Force and other agencies in the development of reuse strategies and their adoption since 1991. CARDS considers reuse an integral factor in creating mature software engineering organizations and practices within DoD. CARDS has demonstrated its skills by applying a wide variety of software methods used by DoD and industry communities. CARDS has also shown an in-depth understanding of the paradigms, processes, and methods needed to make reuse work in practice.

The CARDS program is an Air Force-sponsored program dedicated to furthering DoD and government agency objectives of widespread institutionalization of systematic software reuse into software

acquisition, development, and maintenance. The CARDS mission is to develop a knowledge base of reuse products and processes and to perform technology transfer to other government organizations. The CARDS program is structured around the key elements of the DoD Software Reuse Initiative (SRI) Vision and Strategy (forerunner to the DoD Software Reuse Initiative Program). At the highest level, the DoD Vision for Reuse is "*to drive the DoD software community from its current 're-invent the software' cycle to a process-driven, domain-specific, architecture-centric, library-based way of constructing software.*" Within this vision, planned reuse becomes an essential facet of each software development life cycle phase. CARDS has four main goals targeted to the Vision and devised to identify processes to support the customer-focused goals of:

- Being a premier resource for reuse knowledge which can be applied to improving policy and legal, acquisition, and engineering practices to support software reuse;
- Being a premier resource for C2 knowledge and components;
- Investigating and developing *"advanced"* reuse tools and techniques; and
- Performing technology transfer through a comprehensive reuse adoption strategy tailored to each specific organization's needs.

CARDS advocates the product-line approach supported by a systematic reuse-based systems engineering discipline. As discussed above, a product-line is a collection, or family, of software systems with similar or overlapping functionality and a common architecture that satisfies that set of system mission requirements. A product-line approach involves the development and application of reusable assets (technology base), the development of systems from a common architecture, and large-scale reuse of high quality assets. As your program moves from reuse awareness and understanding to a fully integrated reuse infrastructure, CARDS can provide the experience, lessons-learned, and technology at any point in your reuse efforts to ensure a smooth transition.

The CARDS program provides a comprehensive set of services: domain engineering, library tools and processes to assist with asset management; business and legal advice to ease the establishment of widespread reuse; and consulting/analysis services to better prepare an organization for the advent of reuse. CARDS has a repository of components and demonstration tools accessible on the Web *[see Volume 2, Appendix B for information on how to access CARDS on*

the Web]. For instance, a demonstration of UNAS *[discussed in Chapter 10, Software Tools]* is available to help developers in determining whether it is applicable to their specific effort. CARDS also has hand-in-hand partnerships to assist reuse by key government organizations in the Air Force, Navy, and DoD. With the **Reuse Partnership Project**, technology transfer is facilitated by identifying government organizations that can benefit from interaction with CARDS. CARDS **reuse adoption handbooks** include:

- ***Direction Level Handbook*** assists top-level managers (e.g., Program Executive Officials) in implementing software reuse within a given mission area.
- ***Acquisition Handbook*** assists program managers, contracting, and legal professionals during the contracting/acquisition/ maintenance phases of the software life cycle.
- ***Engineer's Handbook*** assists software engineers and other technical personnel by providing reuse development methods, techniques for reuse integration within their own software engineering processes, and support to the acquisition life cycle.
- ***Component Developer's Handbook*** provides a technical basis for creation of components and tools for domain-specific reuse libraries.
- ***Tool Vendor's Handbook*** assists commercial tool vendors in developing tools to support the reuse process.
- CARDS library operation and maintenance documents.
- Training and educational materials. *[How to obtain these CARDS products is discussed below.]*

The handbooks are comprehensive and should be consulted by all team members. For instance, the ***CARDS Acquisition Handbook*** should be read by all program management personnel. It provides a business framework for incorporating domain-specific reuse into the acquisition life cycle for all major defense systems. It contains guidance on how to encourage reuse, beginning with the establishment of an **Acquisition Strategy Panel** (Team) and the development of an **Acquisition Plan** *[discussed in Chapter 12, Strategic Planning].* It guides in writing the RFP and managing the development effort, as well as follow-on support. It contains recommendations, techniques, and methods for implementing various reuse strategies. The implications and effects of software reuse on the technical, management, cost, schedule, and risk aspects of a program/system during the acquisition and contractual processes are also covered.

While initial CARDS efforts have focused on the command center domain, CARDS also supports other domain libraries. CARDS serves as a testbed for emerging technology and provides a framework for developing and testing other model-based domain-specific reuse libraries. ***CARDS training courses are key to successful adoption of reuse in your program.*** A training plan and system/software engineer's course (how to integrate library-assisted, domain-specific reuse into your program's software development life cycle) is recommended for all personnel. These courses focus on the education of software professionals and support the elimination of cultural barriers to reuse.

Portable Reusable Integrated Software Modules (PRISM)

Sponsored by the ESC and in cooperation with CARDS, the **PRISM** program has established a **Command Center Store** and a model-based reuse library, services, and resources to support the rapid creation of command center capabilities. PRISM was founded on the premise that reuse should be architecture-driven. PRISM provides user-defined rapid prototyping, a generic command center architecture, and a repository of pre-qualified components. It contains existing command center implementations (with 80% functionality) and greatly reduces the time required to develop and field command centers while improving system quality.

Defense Software Repository System (DSRS)

The **DSRS** program was established to develop common technical and management reuse solutions across the DoD MIS domain. Operationally, it provides DoD MIS contractors and government personnel a user-friendly way to acquire reusable software assets. It has a full-service library that contains a high volume of assets having passed a formal certification process to ensure quality. The library contains requirements, design specifications, architectures, design diagrams, source code, documentation, and test suites. The repository uses a domain-based, architecture-centric approach for cataloging and evaluating assets. In addition, it uses domain analysis to identify reuse opportunities and high-demand assets and promotes a consistent, coordinated approach to reuse.

Electronic Library Services and Applications (ELSA)

The **ELSA** project *[formerly AdaNet]* is the operational part of the **Repository-Based Software Engineering (RBSE)** program. The RBSE is sponsored by NASA and dedicated to introducing and supporting common, effective approaches to designing, building, and maintaining software systems by using existing software assets stored in a specialized library. In addition to operating a software life cycle repository, RBSE promotes software engineering technology transfer, academic and instructional support for reuse programs, the use of common software engineering standards and practices, and interoperability among reuse libraries/repositories. *[See Appendix A for information on how to contact the ELSA project.]*

Interoperability Among Software Reuse Libraries

Trilateral interoperability between STARS ASSET, Air Force CARDS, and the DSRS libraries became operational in June 1993. A distributed file system transfers assets among the repositories and allows users transparent access to each library system. This approach was implemented in preparation for the expected growth in domain-specific reuse libraries and is a move towards a large *virtual* library system.

DSSA ADAGE Program

The **Domain-Specific Software Architecture (DSSA)** program was conceived by ARPA as a proof-of-concept for STARS megaprogramming. The DSSA **Avionics Domain Application Generation Environment (ADAGE)** program, which started in 1991, is a joint industry/university research effort designed to apply leading edge basic research to real-world problems.

The DSSA ADAGE approach to component-based development hinges on the creation of a reference architecture — a generic high-level design — for integrated avionics systems. This research has concentrated on creating generic (domain-specific software) architectures and parameterized components that can be used throughout the development process. The goals of the DSSA ADAGE project are to design, document, and develop a set of avionics domain-specific software architectures, avionics software models, a language for their representation and composition, tools for recording avionics knowledge, and tools will enable the rapid development of requirements and software for avionics functions.

DSSA ADAGE aims to free software and system designers from the routine aspects of avionics development, thus giving them time to concentrate on engineering the best solution for their customers. The strategy for reaching this goal is to provide system designers with large configurable components which are easily adaptable to various applications, within an architecture that is open for new functionality.

The DSSA ADAGE architecture-based development process is patterned after Boehm's spiral development model *[discussed in Chapter 3, System Life Cycle and Methodologies].* It adds the concept of using architectures and components to aid in the rapid construction of prototypes and systems, thus reducing the development cycle. One critical risk in prototyping is the amount of effort required to convert the prototype into a production-quality system. Wherever possible, architecture-based development reduces this risk by using the same components for the prototype as for the production system. In addition, it uses historical data and formal semantics of architectures and components to quantify the completeness, correctness, and risks in each stage of a system's evolution.

Avionics domain analysis has shown that developers cannot rely on analytical methods to demonstrate that algorithms or equations will meet requirements. Therefore, developers must test the system through a series of simulations — from simple models and scenarios, progressing through a series of increasingly precise simulation environments, and ending with flight test. A critical feature of an effective avionics development process, therefore, is the ability to rapidly analyze, develop and modify versions of the avionics system (from early prototypes to fielded systems).

The initial DSSA ADAGE domain analysis resulted in the specification of high-level navigation, guidance and flight director functional architectures, as illustrated in Figure 9-15. The capabilities required for providing aircraft flight path management were assessed and allocated based on the functional definitions. This architecture was then compared against other published avionics architectures. The results indicated that this architecture is relatively standard across the industry and hence a likely candidate for consensus building.

Several benefits of rapidly reconfiguring large components are expected. Costs and schedules can be reduced because the architecture provides the high-level software design while the components provide the detailed design and implementation. Thus,

large parts of the design and implementation phases of a program can be eliminated. Parts of the system design can be validated earlier in the life cycle by using the same software for simulation and production. Fewer errors will be inserted into the reconfigured software since there is no hand translation of detailed algorithms from requirements to code. The potential for customer dissatisfaction with the final product can be reduced because initial subsets of the system are built rapidly for customer validation. Finally, the architecture provides a long-term baseline upon which engineers can design improved algorithms and new features. [TRACZ94]

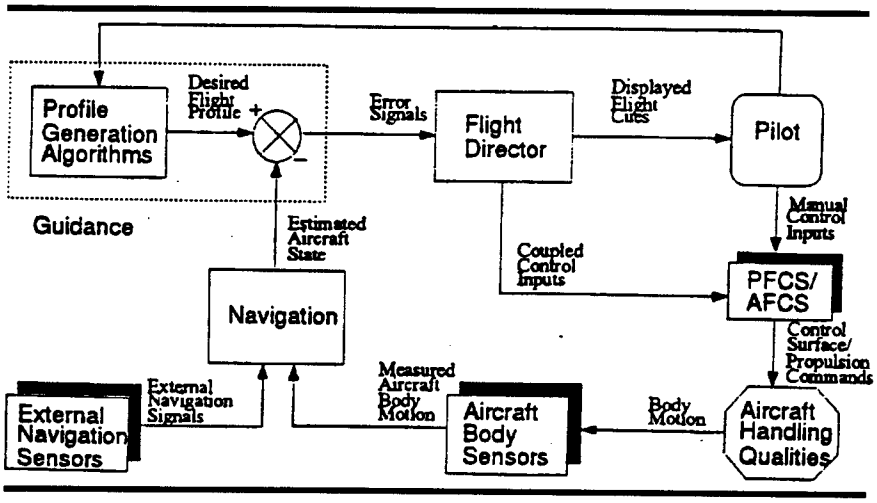


Figure 9-15 High-Level Architecture View of DSSA ADAGE

RICC

Reusable Integrated Command Center (RICC) is a methodology developed by the USSPACECOM and the AFSPACECOM to reverse soaring cost trends in software development and maintenance. The RICC approach builds C2 systems more quickly, cheaper, with higher reliability and maintainability. Cost and performance problems are management concerns the application of technology (Ada) alone cannot cure. While Ada provides a way to solve low-level problems (e.g., by facilitating reuse), the RICC approach identifies underlying management deficiencies causing software development problems. RICC examines how success is measured in development organizations to gain an understanding of the life cycle implications management has on software development. [FRASER93] *[For more information*

about RICC, contact SM-ALC/TIEFA, Building 618, 4235 Forcum Avenue, McClellan AFB, California 95652-1504.]

ADDRESSING REUSE IN THE RFP

When preparing your RFP, ***you should require that offerors propose software architectures that facilitate reuse*** — both internally to the instant software development and from external sources. Offerors should be made aware that there will be rewards (both in source selection and after contract award) to the extent that they can shorten schedules and reduce costs through reuse. This can be accomplished through source selection criteria, special performance incentives (including award fees), and by providing access to **GFS**. Performance and fee incentives based on reuse are not only methods for promoting reusable software, but are also ways for ensuring the offeror — once brought onboard — follows through with the plans for reuse submitted in their proposal. Several acitivities should be performed to plan for reuse during acquisition:

- Technical input to the Acquisition Strategy Panel (ASP) should address reuse.
- Systems engineering trade studies should be conducted to identify and quantify alternative reuse approaches.
- Dialogue with the user should be promoted to assure a balance between requirements, cost, schedule, and reuse.
- If existing, reusable software meets requirements, obtain it. Reuse of existing software saves money and improves reliability by using already proven software assets.
- If existing software requires some modification before it can be properly integrated with your system, analyze the cost/benefits of doing so. Be aware, the cost to modify existing software can sometimes be greater than the cost to develop equivalent new software from scratch. [PRESSMAN92]
- Encourage the use of COTS, GOTS, and common tools within the SEE.
- Encourage reuse of common components by contractors and across multiple contractor teams.

NOTE: See MIL-STD-498, Appendix B, for guidance on incorporating the requirement for reusable products in you RFP.

Be aware that mandating specific approaches to reuse or directing the use of specific reusable assets introduces program risk, and is inconsistent with current acquisition reform initiatives intended to eliminate "*how to*" requirements. ASPs and resulting RFPs must consider the risks and benefits of reuse. For embedded weapon systems reuse should be promoted and implemented at the appropriate level. Some domains may be able to exploit reuse at the software level, but reuse of entire systems or subsystems has the potential for greater pay back while still supporting the objective of delivering world-class, reliable software that costs less to develop and maintain.

If you determine that extensive software reuse is not feasible for your immediate acquisition, you should carefully consider the cost, schedule, and technical benefits/risks associated with requiring that your contractor use a product-line approach that facilitates reuse in future DoD applications. Your RFP should state that designing for reuse is a desired program objective and that a solution based on 100% newly developed code is not. In addition, ***offerors should be encouraged to propose the use of COTS for cost effective satisfaction of as many upfront requirements as possible.*** If the total solution cannot be accomplished with COTS and reuse, extra source selection points should be given to offerors who can evolve the system over time to satisfy 100% of the requirements through the use of COTS. Contractors should be expected to take full advantage of the government software reuse repositories specific to your program domain.

NOTE: The DoD Reuse Initiative Program and the Air Force Software Reuse Implementation Plan should be consulted when developing the SOW, CDRLs, and evaluation criteria for any contract requiring software reuse as part of the software development. Also refer to MIL-STD-498, Volume 2, Appendix D.

A FINAL WORD ON REUSE

The major barrier to reuse, and especially to the use of COTS products, has been the fact that 100% of described user requirements cannot be satisfied in this manner. Traditionally, users have insisted on having 100% of their requirements met — which has demanded custom-developed software.

It is now recognized that many programs for which millions have been invested, which never provided the promised capability, failed for lack of realistic requirements. In fact, programs which have been terminated, some with much notoriety, have been diagnosed in *post mortem* to have been the victims of continually changing or overly precise requirements. With each change the time to delivery is extended, during which environmental changes sufficiently warrant additional changes in requirements. This extends delivery time — resulting in more requirements changes!

The obvious answer is to tradeoff upfront requirements and time. If 80% of requirements can be produced using reusable components, including COTS packages, costs can be reduced by 30-60% and delivery time is frequently reduced by one-half to two-thirds. In effect, this is a form of evolutionary acquisition/development where the initial 80% solution can be quickly implemented, to be followed by refinements (often available from subsequent COTS products or revisions to existing products) which will approach the 100% solution — at a fraction of cost of a totally-customized development.

The best news is that with reuse, instead of the notoriety arising from a *post mortem*, the program manager and user will be congratulated for beating original cost and schedule estimates, and will be cited as innovators who make things happen. ***BETTER TO BE A HERO THAN A HOBO!***

REFERENCES

[AFRIP92] "Air Force Reuse Implementation Plan," Air Force Software Management Division, HQ USAF/SCXS, The Pentagon, Washington, DC, September 18, 1992

[BLUE92] Blue, Dave, "Software Re-Use in Practice: A Reconfigurable F/18 OFP," *Tech-News*, CTA Incorporated, Rockville, Maryland, April 1, 1992

[CALDIERA91] Caldiera, Gianluigi, and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE*, February 1991

[CARDS92] *Direction-Level Handbook Central Archive for Reusable Defense Software (CARDS)*, Informal Technical Report DRAFT-STARS-AC-04104-001099, Electronic Systems Center, Hanscom AFB, Massachusetts, September 24, 1992

[CHRISTENSEN94] Christensen, Steve, "Software Reuse Initiatives," *Lockheed Horizons: Masters of the Code*, Issue 36, December 1994

[FISHER91] Fisher, David T., Myths and Methods: A Guide to Software Productivity, Prentice Hall, New York, 1991

[FRASER93] Fraser, Fred, "Reusable Integrated Command Center (RICC)," White Paper, SM-ALC/TIEFA, July 20, 1993

[GLASS92] Glass, Robert L., Building Quality Software, Prentice-Hall, Englewood Cliffs, New Jersey, 1992

[LENARD92] Leonard, George, as quoted by Lowell Jay Arthur, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993

[LIM94] Lim, Wayne C., "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, September 1994

[NEHRU58] Nehru, Jawaharlal, as quoted by Edgar Snow, Journey to the Beginning, 1958

[PAYTON92] Payton, Teri F., briefing, "Reuse Context," presented at the STARS/Air Force Reuse Orientation, October 14, 1992

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[REUSE92] *Air Force Standards and Guidelines for the Development of Reusable Software*, Headquarters, US Air Force, August 31, 1992

[RSM93] *Reuse Strategy Model: Planning Aid for Reuse-based Projects*, Software Technology for Adaptable, Reliable Systems (STARS) Office, 9-5526, The Boeing Company, Task U03, CDRL 5159, July 31, 1993

[STARS92] Hart, Hal, et al., "STARS Process Concepts Summary," TRI-Ada Conference Proceedings, Orlando, Florida, November 1992

[TRACZ94] Tracz, Will and Lou Conglianese, "An Adaptable Software Architecture for Integrated Avionics," IBM Corporation, Federal Systems Company, Owego, New York, 1994

[VISION92] "DoD Software Reuse and Strategy," Document #1222-04210/40, July 15, 1992

[YOURDON92] Yourdon, Edward N., Decline and Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992

Blank page.

CHAPTER

10

Software Tools

CHAPTER OVERVIEW

Economy of force rightly means, not a mere husbanding of one's resources in manpower, but the employment of one's force, both weapons and men, in accordance with economic laws, so as to yield the highest possible dividend of success in proportion to the expenditure of strength...Economy of force is the supreme law of successful war. —Captain Sir Basil Liddell Hart [HART24]

The key to improving the software development process is to increase productivity through economy of effort. Effort can be decreased by reducing the amount software to be developed through reuse, simplification and refinement of the production process through better methods and procedures, and by automating all manual, repetitive tasks. In this chapter you will be introduced to a variety of software technologies, models of optimum performance, and automated tools that can yield a high dividend of success by harnessing your expenditures of manpower caused by inefficient production processes. Software technologies include: methods, languages, tools, metrics, facilities, techniques, processes, hardware, and other software.

Improving productivity requires making the programmer's job easier. Computer-aided software engineering (CASE) tools provide the means to do this. When properly implemented, tools can increase your chances for program success by eliminating the classic sources of program failure, discussed in Chapter 1, Software Acquisition Overview. Their benefits include the following: they make planning and estimation more accurate; they enhance user involvement; they eliminate errors in requirements and design; they detect and correct defects; they help build flexible, standards-based architectures; and they increase productivity, shorten lead time, and decrease staff turnover through improved job satisfaction.

Despite these benefits, implementing new technologies is an extremely risky business. In fact when poorly planned or understood, "Silver Bullet" technologies are a major source of program failure. To be successful, tool selection and use must be a needs-based, needs-driven process performed with the help of the people who use them. If the tools

do not fit the current process, or if the current process is ad hoc and poorly defined, the training required to learn new processes, or bend old ones to fit a new tool, can be catastrophic. Technology use must be based on detailed knowledge of programmer activities and the software to be built, market analysis of tool availability, cost/benefit analyses, and research into tool evaluations by practitioners and experienced software technology analysts. This implies you must have a technology strategy implemented through a structured, methodical, well-managed Technology Plan.

Throughout this chapter you will be cautioned about common pitfalls to avoid when selecting and implementing new technologies in your program. You will also learn what to look for and who to contact for help. The bottom line in this discussion, however, is that continuous process improvement and advancement to higher software development maturity levels (a Number 1 software management goal) is inextricably dependent on technology insertion. The technologies discussed here include: software engineering tools and environments; program management tools, methods, and models; requirements and design tools; testing tools; documentation tools; re-engineering tools; and configuration management tools.

ATTENTION: **The tools discussed in this chapter are examples of those that have been successfully used on major DoD software-intensive programs. The list is not all-inclusive nor does it imply an endorsement for any tool over another. Because program-specific tool needs vary, it is suggested you contract an organization such as the STSC or the Air Force SSC for information about tools approved for DoD use.** *[See Volume 2, Appendix for information on how to contact these organizations.]*

CHAPTER 10
Software Tools

PRODUCTION EFFICIENCY THROUGH TOOLS, METHODS, AND MODELS

Whatever the system adopted, it must aim above all at perfect efficiency in military action; and the nearer it approaches to this ideal the better it is.
—Rear Admiral Alfred Thayer Mahan [MAHON08]

Software is handmade, therefore, it represents a substantial capital investment. In his book, The Decline and Fall of the American Programmer, Yourdon warns that by the end of this decade the American programmer is going to be as extinct as the dodo bird. Competing in the world software market, he says that high labor costs are forcing the American software industry out of business — the same way the American auto industry was devastated by **Japanese competition** in the 1970's. As illustrated on Table 10-1 (below), our software labor costs have been among the top ten countries with the highest net labor cost per feature point. Yourdon predicts that for American companies to be competitive, they are going to have to go overseas with their software development. In fact, as you read in the *Scientific American* article in the Foreword to this volume, many big American software producers are doing just that.

As illustrated on Table 10-2 (below) by hiring a programmer in India you will pay about 1/10th what you pay an American programmer. The same goes for software engineers and all other software professionals. [YOURDON92] The question is, what do software companies with contracts to develop software for the federal government do who are under federal procurement regulations to hire only American labor? Are they, as Yourdon claims, going to follow suit with our defense aerospace industry which is swiftly losing its market share?

COUNTRY	NET PROJECT COST per FEATURE POINT (U.S. dollars)
1) Japan	$1,600
2) Sweden	$1,500
3) Switzerland	$1,450
4) France	$1,425
5) United Kingdom	$1,400
6) Denmark	$1,350
7) Germany	$1,300
8) Spain	$1,200
9) Italy	$1,150
10) United States	$1,000

Copyright© 1991 by SPR All Rights Reserved

Table 10-1 Countries with the Highest Net Cost per Feature Point Produced

Wait a minute, Yourdon. Is there any other answer to our dilemma? What did the US automobile industry do to turn around business when sales hit rock bottom, and a major segment of the American labor force was relegated to the unemployment lines? They became *leaner-and-meaner* by being more efficient. They invested in new technologies and automated all the production processes they could. [KENNEDY93] They built quality into their products by focusing on the complex values and needs of their customers. They invested in their workforce by training them to be technicians instead of wrench-turners. [REICH91] They increased productivity, reduced their overhead, and lowered costs. They gained control of their process by following the advice of Brigadier General S.L.A. Marshall, who said:

> *Half of control during battle comes from the commander's avoiding useless expenditure of the physical resources of his men...* [MARSHALL80]

COUNTRY	NET PROJECT COST per FEATURE POINT (U.S. dollars)
1) India	$125.00
2) Pakistan	$145.00
3) Poland	$155.00
4) Hungary	$175.00
5) Thailand	$180.00
6) Maylasia	$185.00
7) Venezuela	$190.00
8) Columbia	$195.00
9) Mexico	$200.00
10) Argentina	$250.00

Copyright © 1991 by SPR All Rights Reserved

Table 10-2 Countries with the Lowest Net Cost per Feature Point (US 1991 dollars)

The most efficient way to build quality in, increase customer satisfaction and confidence, and improve your product, is to avoid useless expenditure of manpower and to remove the greatest source of poor quality, ***human-error***. Eliminating the human from your process is the best way to improve your process. When you can reduce a task to a routine activity, ***automate it!*** Automating the software process does away with manual labor and reduces human error and improves productivity. A rule of thumb is: ***more productivity savings come from eliminating the source of error than from humans performing tasks more efficiently***. [HUMPHREY90]

By progressively automating more and more of those tasks performed manually, not only will you increase productivity, you will free your software professionals from their current manual drudgery. They will be rejuvenated for more intellectually rewarding work. They will have the time to find better ways to characterize software objects, to design better processes for controlling and relating them, and more powerful ways for their manipulation. With the power of an automated

environment, you will free the imagination of your development team. Because automation promotes reuse, they will be able to capitalize on the work of others — our greatest opportunity for significant productivity and quality improvements. Automation also improves the way software is supported, and should be used in your software strategy to build supportability in new software product-lines.

CAUTION! Having a defined, mature software development process is a fundamental prerequisite for successful technology use. Having an *ad hoc*, poorly controlled process almost guarantees failure.

Tool Process Improvement Benefits

Computer-aided software engineering (CASE) tools are the most cost-effective, efficient way to increase productivity and product quality. Tools help us efficiently build relatively defect free, easy-to-modify, quality software. Because tools accelerate the pace of development, user requirement, schedule, and budgetary constraints are easily accommodated. Tools also aid in alleviating the classic reasons why software acquisitions fail *[discussed in Chapter 1, Software Acquisition Overview]*. Aggarwal and Lee restate the most common reasons for software acquisition failures:

1. **Improper planning and estimation** because the system is too complex and no scientific methodology exists to correctly estimate resource requirements;
2. **Lack of user involvement** resulting from behavioral, organizational, and political issues;
3. **Inadequate requirements analysis and design** due to system complexity, size, and/or misunderstood user requirements;
4. **Inflexibility of design** which causes flow down of defects inserted when modifying one module to interrelated other modules;
5. **Long lead time** where user requirements or technology dramatically changes before system completion, rendering it obsolete before fielding; and
6. **Turnover of personnel** resulting in lack of program continuity, loss of skills and system knowledge, and the introduction of design changes by new personnel not familiar with the program.

Tools alleviate the sources of these problems in the following ways:

1. **Planning and estimation**. Parametric models provide quantitative measures for estimating resource requirements. An example of an estimating model, based on historical data, is one that assumes development effort and cost are functions of software size, the level of technology used, and the time spent in the analysis, specification, and design phases, as illustrated in the following equations:

   $$DevelopmentEffort = \frac{SystemSize}{(Techno \log yCons \tan t)} \div DevelopmentTime$$

   [TOBIN90]

   $$DevelopmentCost = DevelopmentEffort \bullet LaborCost$$

   [RAJA85]

   Structured systems analysis and design tools decompose the system into a hierarchy of logical components. Interfaces among hierarchical modules are precisely defined and module output is tested to ensure desired results. Problems identified at intermediate stages are easier and less costly to correct than those discovered later in development.
2. **User involvement**. To ensure user involvement, they must participate in prototyping, design specification, and standardized data model creation. Prototyping and code-generation tools that include screen painters and report generators allow users to more clearly define their needs. A part of the system can be built which the users can experience hands-on. Seeing in advance how the system will look and act, users can ask for modifications early on. Increments can be tested and improved before the total system is delivered which improves the probability the system will be used.
3. **Requirements analysis and design**. Structured analysis and design, prototyping, and requirements specification tools help the user visualize how the development process is to take place. Process decomposition diagrams, normalized data models, and entity-relationship diagrams aid users and designers in examining and improving system design. Errors or inadequacies are more easily detected on an automated diagram than in textual descriptions. In an automated environment, more time is spent during the analysis, specification, and design phases, reducing the risk of program failure.
4. **Architectural design flexibility**. Rather than waiting until PDSS where little money is budgeted for changes, an automated tool environment promotes open systems architectures which lead

to flexible specifications which are not frozen until late in the design phase. Design tools, prototyping tools, code generators, and a repository of well-tested modules enhances responsiveness to user needs. In a modular design environment, the rate of change does not exceed the rate of progress.

5. **Lead time**. Well-tested modules, prototyping, and code generators improve productivity and shorten lead time. Because design tools decompose complex systems into small, easy-to-manage modules that interact only minimally, small, independent teams can perform development tasks in parallel. Module integration is facilitated because interfaces among modules are clearly defined. Through a central repository, the same data models can be used and reused by different teams. By reusing pretested designs, data models, specifications, and code, approximately 75% of the resources needed for development from scratch can be reduced. Productivity is increased and defects decreased.
6. **Staff turnover**. Tools help reduce staff turnover and absenteeism through better job satisfaction. By automating boring, manual tasks, teams have more time for creative thinking and process improvement efforts. The time spent hiring and training new staff members is also reduced. [AGGARWAL95]

Table 10-3 summarizes how tools can be used to improve the development process and eliminate classic sources of program failure.

MANAGING NEW TECHNOLOGIES

The term ***"technology"*** involves many elements when applied to software. It can include methods, languages, tools, metrics, facilities, techniques, processes, hardware, other software and/or anything *non-human* used in the production or support of a software-intensive system. Major technology transitions often involve changing or upgrading the fundamental components of the software development process. Managing new technology requires an understanding of the evolution of software engineering from an historical and predictive perspective. In the Foreword to this volume, "Software's Chronic Crisis," *Scientific American*, Shaw's comparison of the evolution of software engineering to chemical engineering is cited. Shaw explains that both fields evolved from craft to commercialization to professional engineering. While both fields strive to design processes to create safe, defect-free products as fast and cost-effectively as possible, they differ in their approaches. Unlike software engineering, chemical engineering relied heavily on scientific theory, mathematical modeling,

MANAGEMENT INADEQUACIES	PROCESS IMPROVEMENT GOALS	CASE TOOLS
Improper planning and estimation	• Management with metrics • Continuous process improvement	• Planning tools • Standardized data modeling tools • Structured analysis and design tools
Lack of user involvement	• Continuous process improvement • Quality by design • Deployment of technology • User participation	• Diagramming tools • Structured analysis and design tools • Prototyping tools • Requirements specification tools
Inadequate requirements analysis and design	• Customer orientation • User participation • Improved communication	• Prototyping tools • Detailed design documentation tools • Standardized data modeling tools
Inflexibility of design	• Quality by design • Deployment of technology • Increased productivity • Continuous process improvement	• Repository tools • Structured analysis and design tools • Diagramming tools • Code-generating tools
Long lead time	• Increased productivity • Deployment of technology	• Repository tools • Prototyping tools • Code-generating tools
Turnover of personnel	• User participation • Deployment of technology	• Repository tools • Standardized data modeling tools • Code-generating tools • Prototyping tools

Table 10-3 Tools to Implement Process Improvement Goals and Remove Management Inadequacies [AGGARWAL95]

proven design solutions, and strict quality assurance techniques to progress to professional engineering. Contrastly, software engineering (still a cottage industry) has yet to build the scientific foundation upon which to progress to a professional discipline. [GIBBS94] Utz explains that software engineering has been and will be tracking the following evolutionary stages:

- **Crafted**. All engineering begins in the crafted mode.
- **Tooled**. Engineering technologies progress to a tooled mode with stand-alone tools (eventually becoming loosely linked) that help build components to a specification. This, Utz says, is an advanced crafted stage because a crafter with loosely linked tools is still a crafter.
- **Engineered**. In this stage, entire systems are built to specification.
- **Automated engineered**. In this stage, tightly linked tools and hierarchical methods are used to handle design complexity.
- **Automated intelligent engineered**. In this stage, complex system interactions and interfaces are design and built by an automated engineering environment freeing engineers to concentrate on product solutions.

Moving from one stage to the next higher level requires the desire for improved product quality and productivity. To improve the product and process, there comes a time when engineers must move on to new technologies. When advancing to new technologies, the following should be understood:

- If you delay too long in realizing that it is time to move on to the next engineering stage, motivation will occur through failure to advance.
- If you lead the pack and pioneer new technologies before they are proven, your results will be unpredictable and involve greater risk — but could have positive cost or competitive returns.
- The ideal way to advance is to progress to the next stage just behind the industry's risk takers.

Management of new technology is a critical challenge. As managers, we are often skilled in the present technology, but have a only a limited knowledge of the new one or how to transition it. This makes it difficult to assess risks, costs, benefits, and to plan. Some of us rush to implementation; while others give up when victory is just within reach. We are constantly challenged to manage new technologies because (especially in software) no one can afford to stay with technologies that are obsolete, inefficient, or outside the competitive range. When the present process promises no further gains in quality or productivity, there are fundamental mistakes we make when faced with the decision to transition to new technologies.

1. One mistake is to implement a new technology when none is needed at the time. As discussed in Chapter 16, *The Challenge,* what you really needed is continuous process improvement which is only achieved through measurement and control. If your process is out of control, the introduction of a new technology will confuse the situation with the lowering of already poor productivity. In this case, process improvement will produce the same benefit as the new technology. Continuous process improvement establishes a solid foundation upon which subsequent new technologies can be implemented.
2. Another mistake we make is when there has been continuous process improvement, but gains in quality and productivity have leveled off. Not fully understanding how to plan for, select, or implement the new technology, we make the decision to transition. This results in a net gain that is slightly worse than the old process where extra effort is required to get the new technology properly working. This situation is compounded when we conclude that we

have selected the wrong technology, try again, and select yet another.

3. Additional mistakes we make are:
   - We do not know what we need,
   - We believe what we see in print,
   - We do not have the time or resources to make good decisions,
   - We decide to use what everyone else uses, or
   - It looks like a good deal, so we buy it.

Technology involves the application of science to the business of software development. Implementing a new technology means switching from one state to another. Although transitioning to new technology is different than continuous process improvement, the goals are the same. Both techniques should yield gains in quality and productivity. They must both be well-managed and well-ingrained within the management process.

Figure 10-1 (below) illustrates the pressure for technology change representing the software industry as a whole. For example, quality and productivity are linked to the extent that a crafted technology can move up the curve only to the maximum gain shown by the crafted line. Those organizations using engineered technologies can achieve much higher levels of quality and productivity than those using crafted ones. Individual organizations make discrete transitions to new technologies in response to technology pull or user needs. Successful organizations *manage* technology transitions — rather than being dragged through them. [UTZ92]

Technology Strategy

To guide the use of new methods, tools, and techniques, you need a **technology strategy**. This means you must have knowledge of what is needed, what is feasible, and what is available, proven, and germane to your process. It also requires the development of a methodical, well-thought out plan. For example, the software engineering environment you choose should provide a full set of facilities to enhance the efficiency of your process. Therefore, the environment you select must be easy to use, support customization, have an open architecture, be based on mature standards, and have the ability to encompass databases, process data, link tools, and provide for its own evolution. Prerequisites for establishing a long-term **Technology Plan** include:

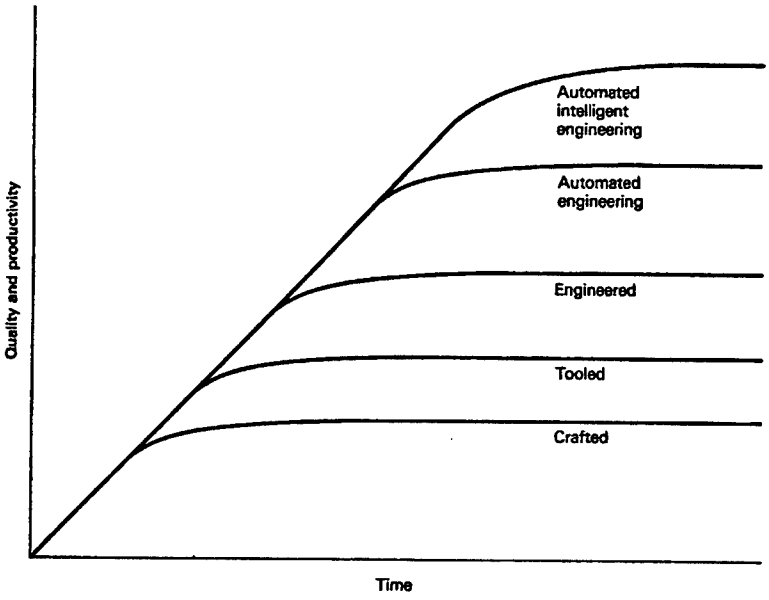


Figure 10-1 Gains in Quality and Productivity Attained by Technology Transitions [UTZ92]

- The establishment of a technology working group,
- Understanding your current technology status,
- Making a systematic assessment of the most promising environments and tools available (by contacting support organizations such as the STSC and the SSC, and by meeting with a wide variety of vendors),
- Developing a common data model for the software environment, and
- Establishing a common user interface so the new tools and facilities are presented consistently to users.

You need to have a structured, methodical, well-managed process for selecting technology or you will wind up with tools that become *shelfware*, tools that do not perform as you thought they would, or spend more on support tools than you do on the system under development. Your Technology Plan must also have economic justification based on thorough cost/benefit/risk analyses. This process can start by creating with a task map of your currently-used technology, and another for that planned. Next, define the resources you are currently expending for each task and assemble a team of cost analysts to determine the resource impact of the new environment and its associated support tools and methods. Each development team's

technology migration plan must be reviewed to assess the accrual rate of programmed savings. Once approved, each team element must commit to their savings schedule. Finally, develop a new schedule based on the estimating factors established, and construct a composite savings schedule for your program as a whole. All teams elements also must agree to this plan. Above all, involve your cost analysts throughout the process, and ensure that they perform their work in detail. It takes time to implement a Technology Plan; but the productivity increases, quality gains, and cost savings are well worth the effort. [HUMPHREY90]

NOTE: See SEI report, *CASE Planning and the Software Process*, CMU/SEI-89-TR-26. *[See Volume 2, Appendices A and B for information on how to obtain SEI reports.]*

Technology Selection

Another big mistake we make is selecting tools before determining how they can support their process. Time and money is often spent buying and learning tools that ultimately get discarded because they do not fit the process. Attempts are also made to make the process fit the tools — which never works. These mistakes happen for several reasons.

Due to DoD procurement process long lead times, tools are often selected before requirements are narrowed down and a process is defined to fulfill those requirements. Another problem specific to DoD is money must often be spent upfront or it is lost. Tools are procured on the assumption that whichever method the tools support can be implemented — usually leading to waste and disappointment. [QUANN93] When selecting new technologies, consider the following:

- **Know your resources** (both positive and negative reports) and use profile sheets.
- **Understand your user's requirements**. Users tend not to fully understand their needs, therefore use an iterative process to extract this information.
- **Use standard evaluation criteria**. Evaluations of unproven technologies are outdated within six months — *Don't believe it unless you see it!*
- **Acquisition involves people, software, and hardware**. Most items are negotiable, and hardware and software vendors

often will loan resources at no cost. Remember, buy only what is needed. Table 10-4 summarizes Mosley's 10 rules for technology selection.

TOOL SELECTION COMMON "CENTS" RULES	
1	Haste makes waste
2	Practitioners are prime evaluators
3	Don't believe it unless you see it
4	Evaluate against user's needs
5	Don't buy it until you've tried it
6	Pay now or pay later
7	Everything is negotiable
8	Buy only what is needed
9	Appoint a tool selection champion
10	Be constructive, not destructive

Table 10-4 Ten Common Sense Rules for Tool Selection [MOSLEY95]

Typical Toolset

A few of the same tools are used in every major software development *[although there are other tools you can use to improve productivity throughout software development].* The **typical** toolset, adequate for development of software of simple to medium design complexity, consists of:

- A **compiler** that translates source code into object (assembly) code.
- An **assembler** that translates assembly code into object code.
- A **linker** that combines the object code output from the compiler and/or assembler together into one executable unit by resolving symbol and address references.
- A **run-time system (RTS)** that provides functions required for execution of Ada programs on a particular hardware platform. Usually, only those RTS functions needed by the specific application are linked into an executable unit, thus reducing code and data sizes.

A more complete Ada toolset should also include:

- A **loader/reformatter** that takes the executable unit generated by the linker, performs any necessary formatting, and loads it onto the target computer.
- A **librarian** that maintains a library of object modules produced by the compiler and/or assembler.
- A **debugger** that provides an interface between the target platform and the programmer to aid in testing and correcting programs.
- An **editor** (ranging from simple text editors to language-sensitive editors) that can display a selected language statement template that is filled in by programmer with program-specific information.
- A **documentation generator** that accepts completed source code, analyzes it, and generates program documentation reflecting the code design. (These are also referred to as **reverse analyzers**.) [ALLEN92]

Utz groups software tools into a series of eight logical toolsets grouped by related attributes, as illustrated on Table 10-5 (below). To be used successfully, Utz says these tools must possess three fundamental characteristics:

- **Simplicity**. The toolsets must support a natural problem-solving process. Toolset functions and the flow from problem to solution must be easy to understand.
- **Standards**. Open systems approaches are breaking new ground in dealing with multi-vendor standardization problems, but the standardization of all system elements is far from imminent. Developing a large software-intensive system involves integrating packages (either COTS, reuse, or custom-built) that must exchange data and support shared functionality. The act of building a new system from these building block components creates complex design problems where they interface — which in turn introduce unforeseen *side-effects* and a new class of defects. Integration problems become enormous when building high-performance, distributed, or networked systems operating on many computer platforms from various vendors, using multiple protocols and multiple database processors. Ideally, tools should support the Ada language and be based on ISO and/or NIST certified standards. Standards are essential for a stable software engineering environment *[discussed below]*.

TOOLSET TYPE	TOOL FUNCTIONS
Development and Test Toolset	• Code creation and editing • Code testing • System/application integration and testing • Change/version control • Configuration management • Optimization • Destructive testing • Performance testing • Other testing as required (functionality, integration, system) • Internal maintenance specification
Definition and Specification Toolset	• Technical analysis of customer's needs • Problem definition • Requirements definition • Structured analysis and design • System/application/architectural specification
Needs and Requirements Toolset	• User's business and environment • International and commercial standards • Market research • Simulation and prototyping • Product positioning
Installation and Build Toolset	• Distribution and software installation • Interconnection • Security • Diagnostic tools • Sizing and tuning (performance) • System/network version control and configuration management
Support Toolset	• Defect analysis, repair, tracking, and reporting • Customer training • Field engineer training
Documentation Toolset	• Documentation plan, to include a list system manuals or product objectives, requirements (including staffing), and assumptions • Requirements definition • Design (including detailed design) • Software module input/output, interconnections, and networking definitions • Testing plans and results • User documentation and on-line help tests
Management Planning and Control Toolset	• Program or product modeling • Program planning • Program estimation, scoping, and costing • Program schedules and milestones • Program quality objectives • Program tracking • Program change, re-estimation, rescoping, and rescheduling • Program integration • Productivity control

Table 10-5 Eight Software Engineering Toolset [UTZ92]

- **Linkages.** Linkages among tools are needed to eliminate unnecessary manual shifting from one tool to the next. For example, the most common linkages are found in compilation, linking, loading, and running tools allowing them to be invoked and used with minimum effort or confusion.

NOTE: See Addendum A of this chapter, "*COTS Integration and Support Model.*"

How the eight software engineering toolsets are combined is illustrated on Figure 10-2. The connections illustrate the concepts of management and documentation performed throughout the development process within an integrated framework. *[NOTE: Although a left-to-right flow is indicated here, it is only used to organize the toolsets in a logical way. It does not imply a left-to-right flow of the software engineering process.]* The data repository is a storage box containing well-integrated support tools, services, and functional databases supported by the engineering process.

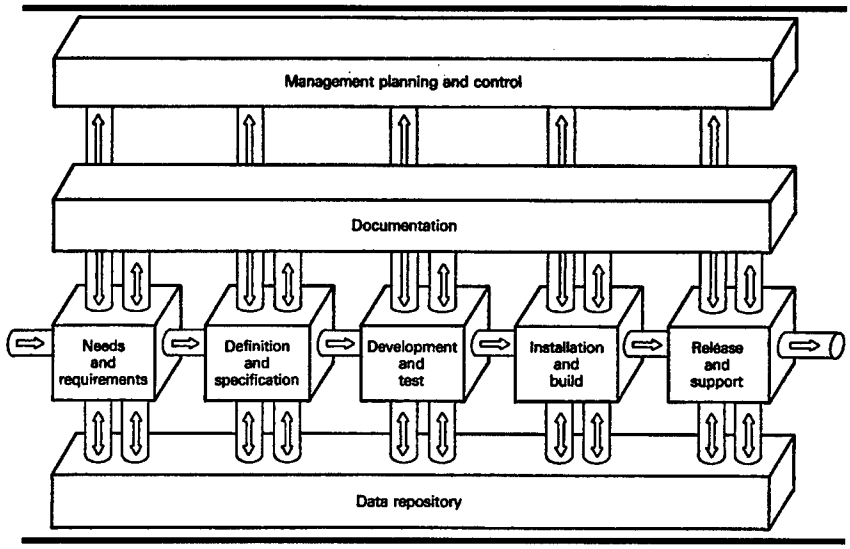


Figure 10-2 Connectivity Among Software Engineering Toolboxes [UTZ92]

Mosley suggests a six-phase process for buying tools and environments, as summarized on Table 10-6 (below). When making tool evaluations, she recommends using SEI publication, *A Guide to the Classification and Assessment of Software Engineering Tools*

TR-10), which lists 140 generic questions for you to consider. These questions can be tailored for specific tools types applicable to the specific development activity you plan to automate. The answers to these question are then evaluated in a simple, weighted, scoring scheme. *[See Volume 2, Appendix A for information on how to obtain this publication from DTIC or the SEI.]* [MOSLEY95]

TOOL SELECTION ACTIVITIES	SELECTION RECOMENDATIONS
Tool Needs Assessment	• Identify the type tool to purchase • Define its required purpose and functionality • Know your price range • Know your hardware constraints
Tool Evaluation	• Read user critiques/evaluations • Visit trade shows, vendor booths, and users' sites • Ask expert users • Identify your essential criteria
Tool Selection	• Obtain objective developer (user) evaluations of alternate tools • Determine which tool(s) satisfy your essential needs • Narrow tool choices to 1 or 2
Tool Acquisition	• Negotiate options and rates • Select the best deal *[NOTE: The "best deal" may not necessarily be the "best choice!"]*
Tool Transition	• Break the tool in slowly • Learn to use essential features initially - Method - Editors - User interface - Data entry - Reporting • Upgrade later to more advanced features - Customized menus - Importing/exporting other tools - Tailored report writing
Tool Vendor Feedback	• Warranty period • Tool maintenance - Problem reporting - Bug fixing - Upgrades

Table 10-6 Tool Selection and Evaluation Activities [MOSLEY95]

More Cautions About CASE Tools

CASE tools are often used to produce graphic representations of systems specifications for the user's review. The difficulty with many CASE products is that they produce abstract graphic representations that can be as difficult for users to understand as traditional specifications. CASE diagrams often help designers communicate

automated diagram generation can have the effect of overwhelming the user with stacks of specifications. Users then become less able to visualize how the system will work, and are less effective participants in its development. Your developer must first decide on a specific development methodology, and then look for those tools that are necessary to accomplish the specific needs that evolve. ***Excessive complexity in the development process is inordinately counterproductive.***

Also be aware, CASE technologies have an associated cost which must be planned and budgeted for which involves much more than the actual dollar cost of the technology. Training costs must be considered and are easy to quantify. The costs that are not so easy to estimate include, but are not limited to, the complexity of the technology (the engineer must, in addition to understanding the software to be built and the implementation domain, learn the complexities of his tools), the cost of tailoring the technology to the development process, and the risks associated with the technology. Be cautioned, (as you learned in Chapter 1, *Software Acquisition Overview*), *Silver Bullet* technologies are a major source of program failure. The great promise of technologies is that they can and do deliver more benefits than their costs — ***but they must be properly planned for and managed!***

NOTE: See SEI reports: *Issues in Tool Acquisition*, CMU/SEI-91-TR-8 and *Guide to CASE Adoption*, CMU/SEI-92-TR-15. *[See Volume 2, Appendix A for information on how to obtain SEI reports.]*

Technology Transition

Most software technology is, in fact, multi-use and transcends all domains. The theoretical knowledge and practical *know-how* that comprise the essence of software technology are more ripe for **transitioning** than the software products themselves. Software technology, developed for one intended use, can often be directly applied to other programs within the domain, and even to other domains. It must be everyone's goal to facilitate the movement of ideas, tools, experience, and knowledge among all members of the software development community. This does not simply mean among DoD practitioners, but also to and from our software development industry partners, our colleagues in the other military services and defense agencies, and academia.

NOTE: DoD experience has demonstrated that the most effective results, particularly at integration, occur when the prime contractor and all software subcontractors use an *identical* software development environment with on-line connections among all developers.

You have the opportunity to encourage the transfer of technology through your software development contractor. The knowledge, tools, and experiences gained on your program must be transitioned to other software developments within DoD. They should also be transitioned to your program through contractor personnel who work on military and commercial assignments within their companies. Civilian demand has spurred industry into rapid technological advances in areas of importance to DoD's mission, such as large-scale communications, networking, expert systems, and other forms of artificial intelligence and telerobotics. [ZRAKET92] Efforts in industry to create better software tools, techniques, and more effective management practices have parallel efforts in universities and government-sponsored laboratories, providing further pools for technology transition.

NOTE: See SEI report, *A Conceptual Framework for Software Technology Transition*, CMU/SEI-93-TR-31. *[See Volume 2, Appendix A for information on how to obtain SEI reports.]*

SOFTWARE ENGINEERING TOOLS AND ENVIRONMENTS

A supermarket full of CASE tools, tool environments, and support services exists for your Ada software development. Tools range from commercially-developed ones to the program-specific ones you develop. Justice cannot be done to all available Ada tools; therefore, only a few selected successful tools are included here. The same is true for tool support services that range from DoD organizations to professional consortiums comprised of government and industry experts and practitioners.

CASE Tools

Successful software development demands the implementation of **computer-aided software engineering CASE** tools. These tools should include some or all of the following capabilities:

- Model user requirements in a graphical format (e.g., data flow diagrams, entity-relationship diagrams);
- Create software design models for both data and procedural code;
- Check defects and consistency;
- Analyze and cross-reference all systems information;
- Build prototypes of systems and enable simulations;
- Enforce development standards for specification, design, and implementation activities throughout the software life cycle;
- Generate code directly from design models;
- Provide automated support for testing and validation;
- Provide support for reusable software components (in the form of designs, code modules, data elements, etc.);
- Provide interfaces to external dictionaries and databases;
- Re-engineer, restructure, and reverse engineer existing software; and
- Store, manage, and report software-related and program management information. [YOURDON92]

WARNING! The implementation of a CASE tool is a complex process the success of which depends on more than having just the right tool with the desired features.

Software Engineering Environments (SEEs)

A **software engineering system** is a collection of all the elements contributing to the development of a software product: people, equipment, data, repositories, procedures, methodologies, and tools. A **software engineering environment (SEE)** is a harmonious collection of integrated methods, CASE tools, procedures, and a management information system for monitoring and control. An **Ada programming support environment (APSE)** is an Ada SEE configured to meet the specific needs of an individual program. Normally contractors propose the SEE. This should be a specific evaluation criterion during source selection. You must be assured, not only of the quality of the SEE — but of the contractor's experience in using it. A SEE also produces information about:

- The software under development (e.g., specifications, design data, source code, test data, and program plans);
- Program resources (e.g., costs, hardware, and software engineering personnel, their assignments, and management duties); and
- Organizational policy, standards, and guidelines on software production.

NOTE: See STSC publication, *Software Engineering Environment*, April 1994, for guidance on how to acquire an SEE and evaluations of commercially-available SEE products.

A good SEE automates or facilitates compliance with identified standards by avoiding cumbersome, time consuming, and expensive manual efforts. It provides and integrates tools that assure consistent application of an effective development process by eliminating design defects early in the process. Such a process makes quality a predictable result. The tools and methods commonly included in a SEE, either dependent or non-dependent on life cycle phase, are illustrated on Table 10-7.

NOTE: Sample RFP paragraphs describing the requirement for a SEE are found in Volume 2, Appendix M. Also refer to "*Reference Model for Program Support Environments*," SEI/CMU-83-TR-23/NIST, Special Publication 500-213, November 1993, for another list of capabilities to include in a SEE.

UNAS

The **Universal Network Architecture Services (UNAS)** was initially developed by TRW for the **Command Center Processing and Display System-Replacement (CCPDS-R)** program *[discussed next]*. It is comprised of a common-layered architecture approach and a supporting suite of reusable components, tools, and instrumentation which represent very high-level language primitives for building distributed C2 systems in Ada. The product, having achieved impressive productivity and quality gains on large-scale, distributed systems, includes support software for rapidly constructing, modifying, and fine-tuning candidate architectures and instrumentation for evaluating the performance and characteristics of alternate approaches, prototypes, and final solutions.

TOOL SELECTION ACTIVITIES	SELECTION RECOMENDATIONS
Tool Needs Assessment	• Identify the type tool to purchase • Define its required purpose and functionality • Know your price range • Know your hardware constraints
Tool Evaluation	• Read user critiques/evaluations • Visit trade shows, vendor booths, and users' sites • Ask expert users • Identify your essential criteria
Tool Selection	• Obtain objective developer (user) evaluations of alternate tools • Determine which tool(s) satisfy your essential needs • Narrow tool choices to 1 or 2
Tool Acquisition	• Negotiate options and rates • Select the best deal *[NOTE: The "best deal" may not necessarily be the "best choice!"]*
Tool Transition	• Break the tool in slowly • Learn to use essential features initially - Method - Editors - User interface - Data entry - Reporting • Upgrade later to more advanced features - Customized menus - Importing/exporting other tools - Tailored report writing
Tool Vendor Feedback	• Warranty period • Tool maintenance - Problem reporting - Bug fixing - Upgrades

Table 10-7 Common Tools and Methods of a SEE [MARCINIAK90]

The UNAS environment solves interface design problems without introducing unnecessary *side-effects* within pre-existing code by automatically generating integration links. It transforms a 4GL input into a 3GL (i.e., Ada) output which then gains measurable machine language performance. It gives the user (developer) the flexibility to experiment with, and change, high-level architecture components throughout the software life cycle. A UNAS user can build the architecture-defined system integration software in a week or less (compared to months if custom code is written by hand). A series of alternative architectures can be quickly built, tailored (non-standardized) for highest performance, and selected. The system integration code can be changed to fit building block component usage requirements, leaving functional packages virtually untouched with system performance uncompromised. A seamless, errorless integration can be achieved with resulting cost and schedule savings.

UNAS has been described as

> *the epitome of "showcase technology" where the process and products are "exposed" to constant observation, in-depth visibility, and constant improvement.* [CRAFTS93]

UNAS can be used with a range of **open systems environments**, including most POSIX-compatible Unix platforms, Rational,™ and VAX/VMS. UNAS also supports platform interoperation so that applications can cooperate in a heterogeneous network while UNAS performs the run-time translation of data formats between dissimilar platforms. [ROYCE91]

NOTE: More information about UNAS can be obtained from the STSC or the Rational Software Corporation. It is also available for inspection and experimentation from the CARDS or ASSET Libraries (discussed in Chapter 9, Reuse). *[See Volume 2, Appendix A for information on how to contact these sources.]*

CCPDS-R Ada Success Story/UNAS Tool Design

The **Command Center Processing and Display System-Replacement (CCPDS-R)** program was designed for use by US Space Command. The CCPDS-R is a highly distributed system with stringent performance and reliability standards that provides strategic missile warning and real-time communications and displays. The program entailed the development of over one million lines-of-Ada code for three physically independent subsystems deployed at various Joint Command sites.

The development team encountered a problem domain consisting of a distributed architecture that was extremely complex to model or evaluate on paper. As the program unfolded, it was realized the architecture would embody much more than just system structure. The architecture became the highest priority during the design stages, as it had to address the majority of the high-leverage design tradeoffs that were driving performance, reliability, and adaptability.

The problems the developers encountered were generally not related to physical theory — nor were they governed by any well-understood form of mathematics. They were more a function of the underlying

platform implementation (i.e., the hardware and operating system) and interdependent system-level characteristics. The team realized they had a challenge but did not want to design the architecture from scratch. This, they concluded, would result in a hand-crafted, custom solution with innumerable system unknowns, ambiguities, and complexities. Anticipating changing requirements and deadlines, a bad architecture would have ultimately resulted in program failure. This was all occurring in 1987, when Ada was only 4 years old. The conventional design tools of the day lacked the technology necessary to produce the architectural solution they needed early in the development. The team decided to design an environment that could define an early architecture while concurrently assessing its quality. The SEE they developed was called "***UNAS.***"

The benefits of combining Ada with CASE technology are well documented on the CCPDS-R program. Of the one million source lines-of-code (SLOC), 40% were reusable. After only 1/3 of the code had been written, the productivity was 40% higher than expected. This was attributable to the use of Ada, in concert with UNAS. (Their original productivity projections were based on historical data gathered from completed Fortran programs.) The CCPDS-R product has proven to be highly reliable. The average change to CCPDS-R's configured software takes approximately 16 hours, based on over 700 changes *[recorded at the time the report upon which this discussion is based]* each of which has affected only about 60 SLOC. These significantly low figures indicate good change localization, due in part to Ada's packaging features and a good flexible UNAS architecture.

Regarding Ada's overall performance, the team found that ***quality costs less with proper use of Ada***. For instance, Ada's strong typing forces early, precise definition of software interfaces, which in turn reduce downstream integration problems and risk. The Ada compiler performs significantly more semantic and syntactic checking than non-Ada compilers, including identification and recompilation of obsolete components. Also, technical reviews and audits of in-process and completed Ada modules is enhanced by a consistent representation format throughout the product design and development.

Lt. General Carl G. O'Berry, former Air Force Deputy Chief of Staff (Command, Control, Communications, and Computers), stated,

> *Ada is an ideal language for an architecture-based environment. It allows designs to be abstract and at the*

same time provides the flexibility for each developer to concentrate on their part of the design. Two widely different programs, CCPDS-R and Cobra Dane, were delivered on time, under budget, and did what the user wanted. They both used the same architecture, the same standards, and the same language—Ada. [O'BERRY93]

With UNAS enabling early software quality assessment, schedule and cost overruns of the past were turned around. Figure 10-3 illustrates the difference between the CCPDS-R code development and integration schedule, compared to traditional schedules. The traditional curve (i.e., a *waterfall* approach) is driven by paper reports and design reviews. This approach delays discovery of architectural design breakage until late in the development process. Late breakage almost always results in program delays and cost overruns, since defects are much more expensive to find and fix when detected and corrected earlier in the process. In contrast, UNAS permits an evolutionary approach by allowing developers to produce a series of demonstrable products/builds, enabling early detection of requirements and design flaws. These early problem resolutions are depicted in the center and left-hand curves as small blips, as opposed to the jagged breaks in the traditional curve. **Walker Royce**, TRW's UNAS development manager, pointed out that, "*With [UNAS's]*

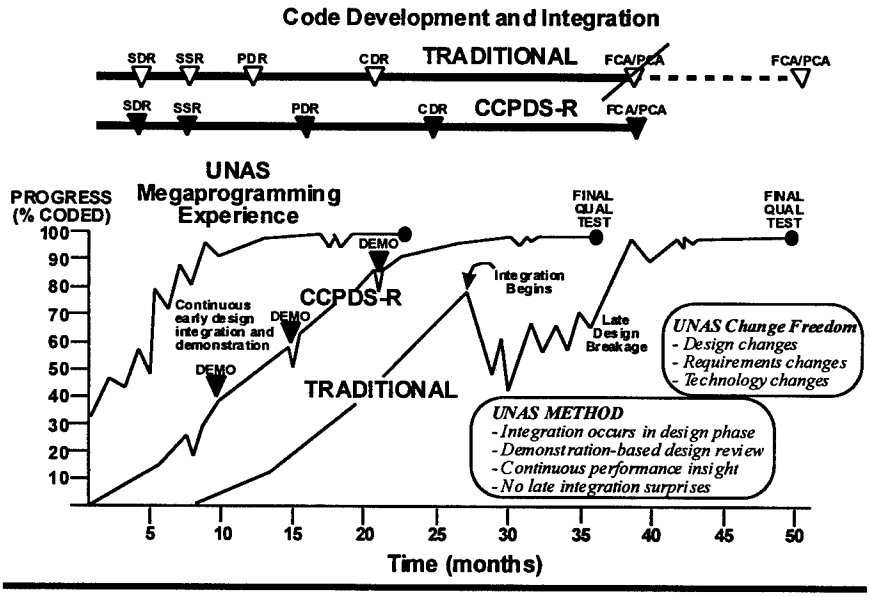


Figure 10-3 CCPDS-R Time Savings Approach

resulting in lower risk and lower costs." [ROYCE93]

The benefits of using UNAS technologies have proven successful on other programs. As Lt. General O'Berry mentioned above, UNAS was also used on the USAF **Cobra Dane** program which required large-scale development of real-time radar software in Ada. It is also being used by Eurocontrol's **Central Flow Management Unit** that manages airspace planning throughout Europe.

Rational Apex™

The **Rational Environment™** is an integrated, interactive SEE for total life cycle control of large-scale Ada programs by enforcing the software architecture throughout development. It also has an Ada-specific editor, an incremental compiler, debugging tools, configuration management, and process and version control facilities. The environment includes features for team support, software testing, large-scale software reuse, and PDSS maintenance. In July 1993, Rational introduced a next generation SEE, **Rational Apex,™** an open-systems implementation of the Rational Environment.™ Its features include:

- **Persistent intermediate representation** enables the capture and management of information about a user's software and stores it in a repository accessible by analysis and design tools, editors, compilers, debuggers, and re-engineering and maintenance tools.
- **Optimal recompilation** recompiles only those individual statements or declarations (depending on the code) that are changed, added, or deleted.
- **Configuration management** automatically keeps detailed records of changes made to the code and prevents unauthorized changes. It enables multiple developers and teams to work from the same baselined code and generates a complete history and program tracking.
- **Rational Subsystems™** control and enforce the system architecture throughout the life cycle and implement iterative development methods that reduce risk by exposing the architecture early. They also aid in identification, packaging, and distribution of reusable components.
- **Compiler independence** allows Apex™ to be used with any Ada compiler, from any vendor, on any platform.

ASC/SEE

According to **Colonel Robert Lyons**, the F-22 SEE is one of the most comprehensive of its kind being implemented by DoD today. He reports that where most environments focus only on the original software development, the F-22 program extends the role of a SEE to the PDSS phase, making it a part of the total weapon system life cycle. The **Aeronautical Systems Center/Software Engineering Environment (ASC/SEE),** used on the F-22 program, addresses the need for automated standards compliance by using electronic development files, documented templates, and management tools customized to standards.

COHESION™ Team/SEE

Digital Equipment Corporation, the developer of the F-22 ASC/SEE, Rational Software Company, and a consortium of 19 other software tool vendors worked together to develop a comprehensive, universal SEE. The **COHESION™ Team/SEE (CTS)** runs on any combination of Digital Unix, SunOS, HP-UX, or IBM-compatible or Macintosh PC operating system, workstation, and server platforms. Working from developer (user) requirements, the team produced an Ada environment populated with *plug-'n-play* tools, technologies, and services conformable to user-specific needs. The CTS' transparent, object-oriented environment manages tool behavior details; thus, it allows users, with little additional training, to *mix-'n-match* new tools and technologies with existing software.

The environment enhances inter-team communication, knowledge-sharing, and development maturity though advanced programming and process management functions. Figure 10-4 illustrates the categories and functions automated within the CTS environment. Browsers and library tools help in visualizing product development, while the graphical workspace paradigm automatically implements those functions needed by the developer to complete his task. Although on-line help and tutorials shorten the user's learning curve, worldwide educational, consulting, and support services are also available to augment in-house expertise. [SHATZ96] *[See Volume 2, Appendix A for information on how to contact Digital about this SEE.]*

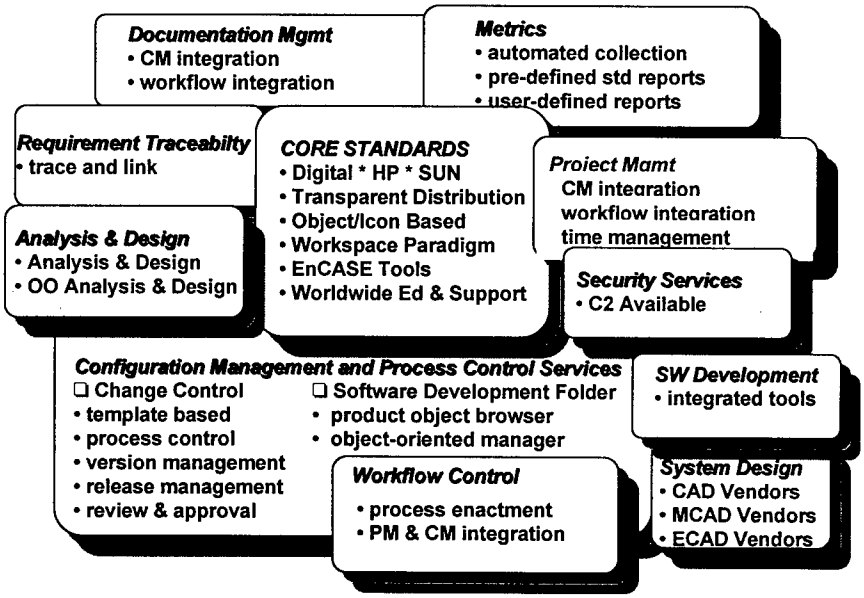


Figure 10-4 COHESION SEE Functions [SHATZ96]

Demonstration Project SEE

The **Demonstration Project SEE** is comprised of approximately 50 workstations connected to three server-class machines. The network is distributed across two geographical sites (over a high-speed link) and is evenly divided between Sun and IBM Unix-based platforms. The SEE is populated with a set of tools that support the desired end-user functionality. The **Air Force Space and Warning Systems Center (SWSC)** uses Ada for their application product-line, and has selected the Rational toolset (Apex, Verdix, RCI, Ada Analyzer, SoDA, and Rose) as key parts of their SEE. In addition, because the application is based on a TRW-originated architectural infrastructure, the SWSC has adopted the corresponding toolset (SALE, RICC Tools) to augment the SEE. Figure 10-5 (below) illustrates the SWSC's emphasis on process technology (the Process Modeling, Program Management, Process Enactment, and Metrics functionality clusters).

Table 10-8 identifies the supplier of each tool shown in Figure 10-5. As shown, the SWSC selected Rational and TRW as major toolset providers for the Demonstration Project. Also shown in the table are four tools developed with STARS support. [RANDALL95] The integration of the SEE tools was accomplished through a combination

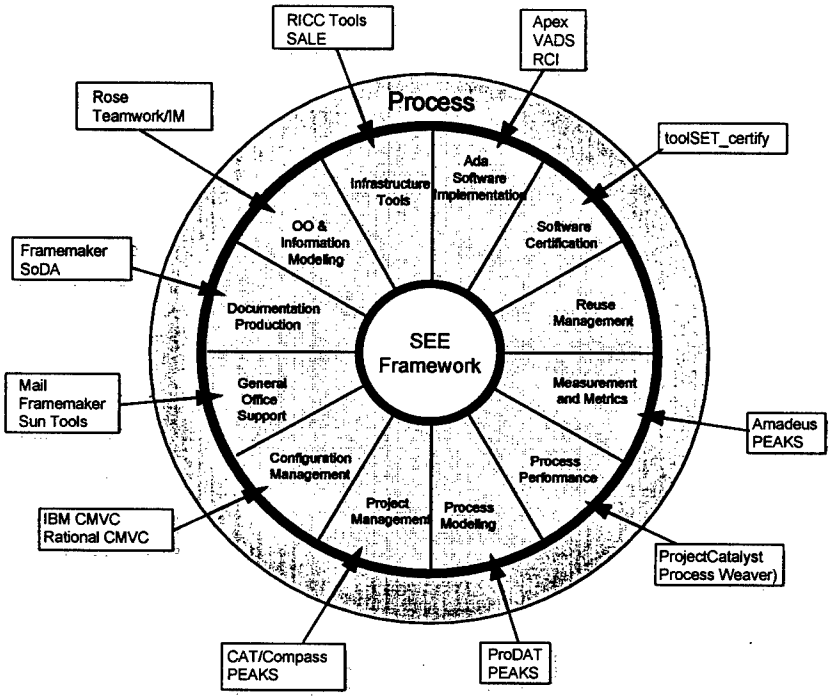


Figure 10-5 Demonstration Project SEE Tool Functionality Groups

of techniques and mechanisms. Control integration (tool invocation and communication) is provided by:

- IBM AIX SDE WorkBench/6000 broadcast messaging service, the Process Weaver message service, operating system process services, and TCP/IP sockets. The WorkBench and operating system process controls provide local service within a user's machine session. Process Weaver and TCP/IP provide service among users and among machines.
- Data integration is provided by the Oracle Relational Database Management System (RDBMS) and the operating system file system.
- Presentation and user interface integration is provided by the XWindow system and the Motif window manager.
- Process integration is provided by the STARS-sponsored Process Support Environment toolset: PEAKS, and ProjectCatalyst (in conjunction with Process Weaver). [RANDALL95]

CHAPTER 10 Software Tools

TOOLSET	TOOL NAME	VENDOR/ DEVELOPER	PURPOSE
Rational Ada Development Toolset	Apex	Rational	Code creation and testing
	RCI	Rational	Interfaces Apex with other Ada compilers
	Rose	Rational	Object-oriented analysis and design
	SoDA	Rational	Automated document generation
	VADS	Rational	Verdix compiler
TRW Architecture Infrastructure Support Toolset	RICC Tools	TRW	Application display, message, and database definition
	SALE	TRW	Application network definition (used with UNAS)
	UNAS	TRW	Application network manager
STARS-Supported Tools	Amadeus	Amadeus Software Research, Inc.	Metrics repository and analysis
	PEAKS	Cedar Creek Process Engineering (ccPE)	Process modeling, planning, and plan simulation
	Project-Catalyst	Software Engineering Technology (SET)	Low-level process definition and process execution
	toolSET certify	SET	SET Cleanroom certification testing support
Other Tools	CAT/Compass	Robbins-Gioia	Program management
	CMVC	IBM	Configuration management and tracking system
	Frame-Maker	Frame Technology, Inc.	Documentation and publication
	ProDAT	Embedded Computer Resource Support Improvement Program (ESIP), managed by AF Air Logistics Center, Sacramento	IDEF-oriented process definition
	Process Weaver	Cap Gemini America	Process workflow manager
	SunTools	Sun	General office support
	Teamwork/IM	Cadre Technologies	Information modeling

Table 10-8 Demonstration Project SEE Tool Suppliers [RANDALL95]

I-CASE

The **Integrated Computer-Aided Software Engineering (I-CASE)** program provides a means for DoD users to purchase standard software engineering environments for development activities throughout the department. I-CASE, to the maximum extent practicable, utilizes COTS hardware and software components for development environments and includes target environment runtime licenses so development activities can deliver complete executable software systems. I-CASE provides automated tool support for all MIS software developments, including development of new MIS applications, maintenance, re-engineering of existing MIS applications, and tools to support the entire software development and maintenance life cycle. I-CASE also provides the support elements necessary to implement, operate, and maintain the I-CASE environment, such as training, maintenance, and technical support.

I-CASE SEE encompasses an integrated information. The repository, critical in improving software production and quality, allows flexibility through the introduction of new standard technologies as they become available. The repository serves as a hub, synthesizing program management, configuration management, and quality assurance into the entire development process.

I-CASE, an Ada-based environment, utilizes evolving standards supported by DoD, including SQL, XWindows end-user interfaces, POSIX, the NIST Reference Model for Frameworks, and GOSIP. New technology can be added as the I-CASE marketplace matures and the I-CASE SPO closely monitors its evolution to make sure it keeps pace. The I-CASE repository data model is commercially available, allowing vendors to build I-CASE easily integrated tools. The I-CASE operational environment is illustrated on Figure 10-6.

The I-CASE acquisition was accomplished on behalf of the **Defense Information Systems Agency (DISA)** for use by all military services and defense agencies. The I-CASE contract was awarded in April 1994 to Logicon, Incorporated. The Logicon SEE provides full life cycle support with an automated workflow control feature that enforces chosen the software development methodologies, automatically collects metrics data, and is easily tailored as processes mature. All program data are stored in a single, integrated repository which is implemented using data bridges to those tools that do not directly support the repository data model. *[See Volume 2, Appendix*

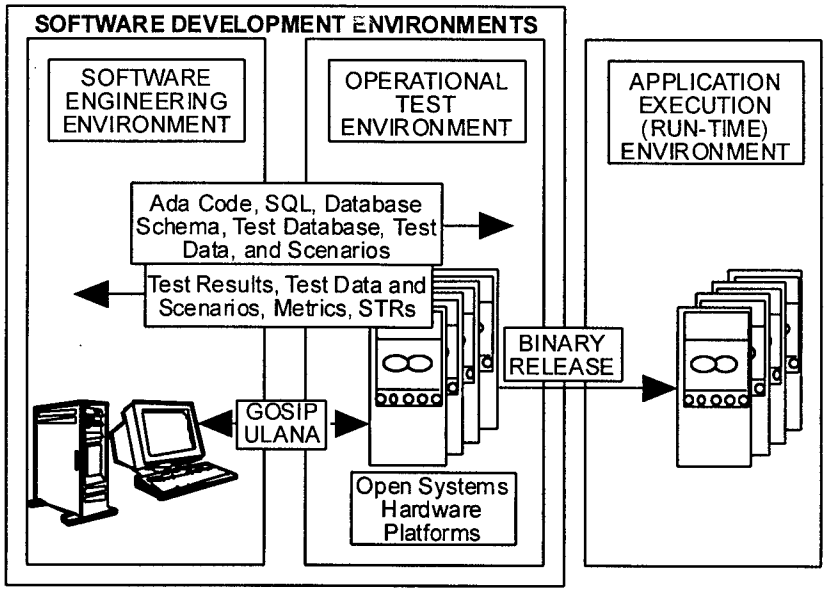


Figure 10-6 I-CASE Operational Concept

L for descriptions of I-CASE tools and Appendix A for points of contact.]

Ada-ASSURED

Ada-ASSURED (designed by GrammaTech) is an easy-to-use Ada development environment that includes language-sensitive editing, standards enforcement, browsing, and pretty printing in a single package that can be integrated with any Ada compiler. It was designed to increase programmer productivity and enforce compliance with accepted Ada programming style rules, including the **Ada Quality and Style Guidelines** developed by the Software Productivity Consortium (SPC) and recommended by the Ada Joint Program Office (AJPO).

Ada-ASSURED automates a variety of tasks so novice and advanced Ada programmers can write and maintain code faster with fewer errors. Its formatting features generate easily readable, understandable, modifiable, and reusable code. Ada-ASSURED is based on attribute grammars which create an internal tree structure. This allows code to be accessed, manipulated, and formatted with greater precision than conventional editors and pretty printers.

Ada-ASSURED follows the SPC's Guidelines wherever possible (e.g., horizontal spacing, indentation, alignment of declarations, alignment of parameter modes, number of statements per line, automatic naming of end statements, and mode indication.) When viewing a file, pretty printing adjusts to window width. If the user resizes a window, the application is reformatted to fit the window, making it easy to read [of course, the user may select a default width (e.g., 80 characters) for printing or saving a file]. Two features, *Untouchable Comments* and *Freeform Text*, allow special handling of comments and regions of code that users want to format manually. Its pretty printing feature can run existing code in batch mode for consistent appearance. It also has a language-sensitive editor which formats code as it is typed in.

Ada-ASSURED's LSE features include textual and structural editing. **Structural editing** allows accurate and efficient navigation with respect to code structure. **Structural selection** enables users to choose any code phrase or subphrase without manually scrolling from beginning to end. Although the user can cut and paste the subphrase, Ada-ASSURED only allows pasting where it is syntactically correct. **Structural search** allows users to search for and edit phrases of a particular syntactic category. For example, it can quickly replace all occurrences of a variable *"i"* while ignoring textual occurrences of the character *"i"* or any non-variable occurrence.

Ada-ASSURED supports the product-line approach to software development. The major objective of this approach is to move from the current development paradigm, where code is nearly 100% written from scratch, to the product-line that focuses on substantial reuse. Ada-ASSURED supports this approach for the following three code categories:

- **New code**. Ada-ASSURED automatically enforces SPC Guidelines as code is written (i.e., stricter coding standards may be enforced through user customization of Ada-ASSURED). Formatting, standards enforcement, and syntax checking are all performed automatically.
- **Generated code**. Ada-ASSURED ensures consistency and performs precise style and formatting code generation.
- **Reused code**. Before code is reused Ada-ASSURED runs it through a batch mode to ensure that it is consistently formatted so it is easier to read and understand — thus, easier to maintain and reuse.

Although Ada-ASSURED provides full textual editing, it can be used with existing text editors. Other productivity features include: context-sensitive templates, hypertext name-based browsing, a user-mappable keyboard, and a scripting language. The latest version, Ada-ASSURED 3.0, supports Ada 95 and Ada 83. Upcoming enhancements include user-definable header templates and HTML links to documentation, the Ada Language Reference Manual, and the SPC guidelines. Ada-ASSURED runs on most workstation environments, including SunOS, Solaris (Sun or Intel), Digital Unix, HP-UX, AIX, and IRIX. Ada-ASSURED's vendor is a strategic partner of the Software Productivity Consortium, which uses Ada-ASSURED to format all examples in the current Ada 95 Guidelines. Ada-ASSURED is being used on the FAA's National Airspace System (NAS), on the Czech nuclear plant retrofit program, on the Sustaining Base Information System (SBIS) program, and on the Navy Seawolf's BSY-2 program. *[See Volume 2, Appendix A for information on how to obtain Ada-ASSURED and points of contact for programs using the product.]*

WARNING! **Because SEEs are not yet a proven technology, you must to be aware of the danger in using them. If the contractor has not previously used the proposed environment on a job of similar type and complexity, there is a good chance it will cause problems with product quality, schedule, and cost. Environment problems, if not caught early, can be fatal to any program, large or small.** [HUMPHREY95]

PROGRAM MANAGEMENT TOOLS, METHODS, AND MODELS

This technology domain involves the principles, methods, tools, and products that aid program managers. There are a variety of tools to assist management, the features and functions of which include the following:

- **User assistance features**. These aid in the learning and operation of program management tools. They can include:
  - Graphical user interfaces,
  - Tutorials, sources of instruction, explanations of terminology or procedures, and on-line help,
  - Automation of program management processes or functions, and
  - User-defined customization of the interface and tool itself.

- **Program scheduling functions**. These aid in accurately modeling and displaying program scheduling. This involves a variety of methods for analyzing the program mathematically and logically, and defining the constraints which effect the implementation of the Program Plan. They also contain methods for scheduling tasks and for defining the program WBS *[discussed in Chapter 12, Strategic Planning]*. They can include:
  - User-defined customization,
  - Forward/reverse scheduling,
  - Method-driven scheduling,
  - Task priority analyses,
  - WBS definitions, and
  - Program evaluation and review techniques (PERT).
- **Resource management functions**. These aid in accurately modeling program resources. They can include:
  - Program baseline definition,
  - Planned versus actual comparisons,
  - Automatic/manual updating, and
  - "*What-if*" analyses.
- **Program and product estimation functions**. These aid in estimating program schedules, resources, productivity, costs, and product quality, size, and complexity. They can include:
  - Critical path analyses;
  - Tutorials, sources of instruction, explanations of terminology or procedures, and on-line help;
  - Early estimations before requirements and development environments are fully defined,
  - Estimations for all phases and activities for the most commonly-used software life cycle methodologies;
  - Adaptability to various programming languages and functions;
  - Software size estimation or help in defining a method to estimate size; and
  - Schedule and support estimation.
- **Program tracking features**. These aid in collecting useful progress and comparison data which can be used to identify possible problems. They can include:
  - Program baseline definition,
  - Planned versus actual comparisons,
  - Automatic/manual updating, and
  - "*What-if*" analyses.
- **Program reporting features**.
  - Customization of program information,
  - Gantt and resource histogram charts,

- Network diagrams (PERT charts), and
- Cost graphs and spreadsheet reports.

NOTE: See STSC publication, Program Management and Software Cost Estimation, April 1995, for evaluations of approved commercial program management tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

Cost/Schedule/Size Estimation Models

The most commonly-used technology for software estimation is **parametric models**, a variety of which are available from both commercial and government sources. The estimates produced by the models are ***repeatable,*** facilitating sensitivity and domain analysis. The models generate estimates through statistical formulas that relate a dependent variable (e.g., cost, schedule, resources) to one or more independent variables. Independent variables are called ***"cost drivers"*** because any change in their value results in a change in the cost, schedule, or resource estimate. The models also address both the development (e.g., development team skills/experience, process maturity, tools, complexity, size, domain, etc.) and operational (how the software will be used) environments, as well as software characteristics. Mosemann tells us that ***"software size estimation is the main cost and schedule driver — with environmental factors adding extra dimensions."*** [MOSEMANN92] The environmental factors, used to calculate cost (manpower/effort), schedule, and resources (people, hardware, tools, etc.), are often the basis of comparison among historical programs, or can be used to assess on-going program progress. Because environmental factors are relatively subjective, a rule of thumb when using parametric models for program estimates is to ***use multiple models*** as checks and balances against each other.

Parametric Model Selection and Use

Although you may use any model(s) you choose, you will need to justify their appropriateness as part of your DAB and/or MAISRC reviews. The best models are those that assess the cost elements relevant to your program and with which your ***cost analysts are the most skilled.*** Experience shows that the best estimates are produced when the models are used by people who fully understand them. The best cost analysts are also those who understand the basic concepts of software engineering and who have reached a thorough understanding

of the software being appraised. Conversely, the estimates produced by these same models are significantly less accurate if used incorrectly by inexperienced personnel who do not understand the software being estimated. **Mosemann** explains that we must be careful making our cost estimates based on current software size estimation methods:

> *We are now dealing with only a first generation of software cost models. Issues of software cost and schedule overruns associated with the use of these models ultimately stem from the inability to estimate software size accurately. This is the Achilles heel for our current software cost estimating models.* [MOSEMANN92]

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on measuring software size.

Another important point to remember is that different models assess different parts of the software life cycle and include different cost elements in the resulting estimate. Therefore, direct, unadjusted comparisons of the results from two models is inappropriate and misleading. Also, existing software estimating models ***do not*** estimate all the costs associated with typical DoD programs. It is critical to understand exactly what costs are included in the estimate produced, as well as those not included. Alternative estimating techniques (e.g., expert opinion, analogy, cost performance report (CPR) analysis, cost estimation relationship/factor analysis) must be used to calculate costs not included in your selected software estimation model(s).

Because similar internal parameter settings may be based on different definitions, the inputs and outputs among models can vary, as well as the algorithms used to convert inputs into outputs. Further, many models include other elements (e.g., data, program management, or configuration control) that may be based on a higher or lower WBS level. Understanding the relationship of these estimates to similar, but higher or lower programmatic WBS elements is important. Without thorough knowledge on how to use each model, there is a high probability that your cost analyst will use the model incorrectly and obtain an incredible estimate. **Formal training** on your selected parametric model(s) is critical for overcoming this problem and for their effective and credible use, the best source of which is from the model's vendor.

BE AWARE! Basing your estimates on models built from data which do not correspond to, or mirror, your anticipated development will produce substantially inaccurate estimates.

While *size* is by far the most significant cost, schedule, and resource driver, other factors have impact. The specific environmental parameters used by the estimation models can only produce accurate results when your **input parameter settings** are chosen in a manner consistent with the model's definitions. Remember, the old axiom applies: "*garbage-in, garbage-out.*" It is, imperative to have a solid understanding of the exact definitions of each environmental factors being used, as they vary among models. Simply inputting the values of your specific parameters (size and environmental) and letting the model calculate your cost, schedule, and resources — is not recommended. This approach relies on the database upon which the model was built, and assumes that the model's database reflects your environment. ***This is a dangerous assumption***, because models are typically based on data readily-available to the developer which can differ significantly from yours.

REMEMBER! While many models use similar terms for similar environmental parameters, the way their associated characteristics are defined can differ significantly among models.

A more credible approach is to calibrate your selected models with actual data from developments similar to yours. This calibration tunes the model to your domain and software type, as well as to your program's environment. Only after calibration can the models produce truly credible estimates. To calibrate models, it is necessary to have good, consistently-collected, credible cost, schedule, and technical data from already completed programs. Thus, past experience is used to justify estimates based on more than pure engineering judgment. For example, alternate cost drivers can be used and *what-if* analysis performed to determine their effect on your effort and schedule estimates. These sensitivity analyses are performed to determine to which parameters effort and schedule are the most sensitive. [MARCINIAK90] ***Do not artificially modify your development environment to fit the model.*** The effect of changing your descriptions to fit the model is to inadvertently estimate the wrong software system, thereby producing an inaccurate estimate.

CAUTION! Do not adjust your data to the model! Always adjust the model to your program's parameters.

There are a few reliable sources of cost, schedule, and technical data that can be used to calibrate your estimation models. They consist of databases containing volumes of information consistently collected over the years on a variety of software developments using different languages, from commercial, command and control, and MIS to weapon systems. The vendors of these databases are also available for assistance in developing software estimates using industry and government-wide comparative productivity and cost information. *[How to access these databases of historic software development information is found in Volume 2, Appendix A. Also, you are encouraged to consult early with the Air Force Cost Analysis Agency, also listed in Appendix A.]*

Accounting for the differences between your program and the historic data is often difficult. New technologies or development methods, not addressed at the time the model was designed, may not be easily taken into account. However, compared to other estimating techniques, parametric models produce credible estimates quickly, with a minimum of effort. Table 10-9 lists several currently available models and their applicability to life cycle phase coverage. *[Volume 2, Appendix A lists the of vendors of these models and how to contact them.]*

Cost Analysis Requirements Document (CARD)

The **CARD,** defining the technical and program baseline, is required for weapons systems and has been requested on several MIS programs. It contains information such as program requirements, acquisition strategy, descriptions of hardware and software, basing and deployment plans, maintenance strategy, and schedules. The CARD establishes a framework within which to ask and answer those questions necessary for successful planning. The **Air Force Cost Analysis Agency (AFCAA)** (one of the primary users of the document) provides guidance on the content and preparation of a CARD.

Air Force Acquisition Model (for Weapon System Software)

The **Air Force Acquisition Model (AFAM)** is an MS-DOS/ Windows application designed to assist experienced and inexperienced personnel in performing acquisition tasks for major weapon systems

	COMOMO	REVIC	SASET	SEER -SEM	PRICE -S	SLIM	SOFTCOST-ADA
System Requirements			✓	✓	✓	✓	✓
System Design			✓	✓	✓	✓	✓
Software Requirements		✓	✓	✓	✓	✓	✓
Preliminary Design	✓	✓	✓	✓	✓	✓	✓
Detailed Design	✓	✓	✓	✓	✓	✓	✓
Code/ Unit Test	✓	✓	✓	✓	✓	✓	✓
CSC Integration & Test	✓	✓	✓	✓	✓	✓	✓
CSCI Testing	✓		✓	✓	✓	✓	✓
System Integration & Testing	✓		✓	✓	✓	✓	✓
Operational Test & Evaluation			✓	✓	✓	✓	✓
Production & Deployment			✓				

Table 10-9 Parametric Models Applicability to Life Cycle Phase

programs and most non-major acquisitions. It uses a graphical display environment with a text retrieval capability based on the acquisition life cycle. It has a breakdown of processes, tasks, and sub-tasks performed throughout these phases, in addition to key acquisition references, standards, pamphlets, guides, and handbooks. *[See Volume 2, Appendix A for a source of more information about the AFAM.]*

Program Management Support System (PMSS)

The **Program Management Support System (PMSS)** provides program managers with the skills, methodologies, and automated program management tools to develop program plans and schedules, track technical, schedule and cost progress, conduct *what if* analyses, verify resource management, manage configuration control, and identify program problem areas. PMSS is a fully integrated turn-key system consisting of three components provides visibility into all aspects of a program.

- The appropriate methodology,
- A robust, automated tool for scheduling, information processing, and specific applications development, and
- Analysts dedicated solely to the task of program management.

Automated tools include program management support applications, information systems hardware, maintenance, supplies, training, documentation, and telecommunications interfaces. PMSS on-site program management support personnel include experienced program analysts, financial/budget/costs analysts, configuration managers and system operators. Typically, the system hardware configuration consists of Hewlett-Packard (HP) 9000 series processors and the HP-UX operating systems software, and a variety of output devices, including printers and plotters. User terminals are primarily DOS-based 486 computers. All hardware items are government owned. This configuration has been installed at over 25 sites throughout the United States and supports over 1,000 users and 44 multiple agency program offices.

PMSS provides the acquisition and maintenance management for Air Force Material Command (AFMC) logistics management systems, the Surgeon General, and other operations at AFMC headquarters and component organizations (e.g., Armstrong Labs, Air Logistics Centers, Air Force Center for Environmental Excellence, Material Systems Group, Standard Systems Group, Air Force CALS Program Office). PMSS supports evolving DoD Standard Systems initiatives, such as the Joint Logistics Systems Center (JLSC) Material Management and Depot Maintenance, the Defense Logistics Agency (DLA), Defense Distribution Systems Center and Procurement initiatives, and the Defense Information Systems Agency (DISA). The automated PMSS tool provides large capacity, real-time responsiveness, and capabilities such as:

- Critical path networking,
- The ability to combine multiple networks,
- A complete and integrated graphics capability including network plots, Gantt, bar, line, histogram, scatter, PERT, and pie charts,
- Open architecture,
- An integrated relational database management system that uses a SQL-compliant query language,
- An integrated 4th generation command and applications development language,
- Modules for planning, scheduling, configuration management, cost reporting, executive information systems, and
- Modifiable model networks and work breakdown structures.

A PMSS system is flexible enough to support classic development approaches, as well information engineering, rapid application development and prototyping, and commercial standard, *best-of-breed* manufacturing/production engineering processes. The ability to see problems early and react, based on informed decision-making, is often the difference between a successful program and one out of control.

One factor contributing to the success of PMSS is the contractor's ability to *apply best commercial practices* such as those used by major corporations (e.g., General Motors, GTE, US West, AT&T and EDS). From a corporate perspective, the contractor providing the PMSS solution must have a broad base of experience in successfully implementing the PMSS processes within large, complex commercial operations. This experience base provides the Government with immediate access to best commercial practices which may be tailored and applied to government operations. PMSS has resulted in many tangible and quantifiable benefits, some of the more notable of which are:

- A cost avoidance over **$6 million** in one AFMC/MSG program office.
- PDMSS-facilitated process improvements in weapon systems maintenance management resulting in an estimated:
  - **10%** to **30%** reduction in aircraft PDM cycle time,
  - **4%** to **15%** reduction in aircraft PDM direct labor hours,
  - Savings of **$220 million** (NPV) in over 6 years, and
  - **5:1 ROI**.
- PM/CCAT-facilitated process improvements in aircraft modification management resulting in an estimated:
  - Modification cost reduction,

  - Reduction in aircraft downtime,
  - **$10 million** annualized cost savings, and
  - **10:1 ROI**.
- Yearly savings in excess of **$1 million** through the use of specifically-tailored PMSS applications.

Examples of how PMSS has been successfully implemented are the PDMSS and PM/CCAT, where PMSS is at the heart of each application.

- The **Programmed Depot Maintenance Scheduling System (PDMSS)** module has been deployed and implemented by the JLSC Depot Maintenance Directorate at 21 major defense depot/ maintenance centers. PDMSS, a standard migration system, supports Air Force, Army, Navy, and Marine Corps mission-critical weapon systems depot maintenance operations. PDMSS provides depots with the capability to define, plan, estimate, schedule, budget, status, forecast and measure work performance on major end item maintenance, modification, and overhaul programs. The functions performed by the PDMSS support the business processes associated with major end item maintenance. Its program management methodology is adapted to each depot's workload to support the diversity of requirements and processes. PDMSS provides continuous, uninterrupted, mission-critical support in the maintenance of over 40 major weapon systems. These weapon systems include, but are not limited to:
  - Air Force aircraft,
  - Army helicopters and combat vehicles,
  - Navy aircraft ships and submarines, and
  - Marine Corps weapon systems.
- The **Program Management/Configuration Control and Tracking (PM/CCAT)** module has been implemented by the C-130 System Program Directorate (SPD) at Warner Robins Air Logistics Center. PM/CCAT supports mission-critical modification management and provides total asset visibility of C-130 fleet maintenance. PM/CCAT provides the means to integrate and execute multiple modifications in a single, consolidated schedule. PM/CCAT's integrated modification scheduling process integrates the following information:
  - Aircraft inventory,
  - Designated time-compliance technical orders (TCTOs),
  - Top-level TCTO Data,
  - Aircraft kit availability,
  - Kit delivery schedules,
  - Contract field teams, and

- Aircraft mishap data and PDM schedules.

- PM/CCAT is integrated with PDMSS at Warner Robins Air Logistics Center. This integration provides the SPD and the Air Combat Command (ACC) with asset visibility, current status, and forecasts of aircraft undergoing PDM. Baselining the cost, schedule, and technical content of a program and measuring performance against those baselines, provides the means to assess the pulse a program so a manager can react quickly when problems are encountered. The three goals of the PMSS are:
  - Institutionalizing a program methodology based on planning, baselining, and performance measurement;
  - Facilitating methodology implementation; and
  - Providing program management offices with on-site contractor advice, assistance, and training in the PMSS processes. *[See Volume 2, Appendix A for a source of additional information on the PMSS.]*

ADARTS®

The **Ada-based Design Approach for Real-Time Systems (ADARTS)** is a systems and software design method for designing large, complex systems, including real-time, multiprocessor and distributed systems. ADARTS applies decomposition principles and object-oriented technology to guide systems and software engineers in creating designs which address concurrency and are resilient to change. Starting from a requirements specification, engineers use ADARTS to develop a detailed design that can be mapped either to an Ada architecture or a C+ + architecture [using the Software Productivity Consortium's (SPC's) CDARTS®]. ADARTS supports design for reuse, evolution, and maintenance by managing the impact of potentially critical changes in requirements or designs.

ADARTS provides detailed criteria for designing complex systems, in particular real-time systems. ADARTS is composed of a comprehensive set of activities, each of which contains entrance criteria, heuristics and guidelines, and exit and evaluation criteria. The outputs of these activities produce different architectural views of the system, which capture a particular structure and set of relationships among components of a system or subsystem. Focusing on a different set of concerns, different view allows the engineer to concentrate on a specific, critical set of system/ subsystem properties of the.

The SPC has a complete ADARTS support package that includes a detailed guidebook, case studies, training courses, on-going method improvements, video presentations, ADARTS automation support through commercial software tools, and extensive consulting services. ADARTS has been selected as the standard design methodology on several major Ada development programs, including the **F-22 Advanced Tactical Fighter** program. In addition, ADARTS (and other Consortium technologies for requirements engineering and Ada programming) has been used on key programs such as the **EF-111A System Improvement Program** and the **C-130J Hercules program**. *[See Volume 2, Appendix A for information on how to contact the SPC Clearinghouse and ADARTS support line. ADARTS is a service mark and CDARTS is a trademark of the SPC.]*

Process Weaver®

Process Weaver® (a registered trademark of Cap Gemini Innovation) is a process enactment, work flow management tool which improves productivity as well as quality. Process Weaver® frees developers from process concerns so they can devote more attention to development tasks by enacting (i.e., simulating) development and program management processes within the software engineering environment (SEE). Process metrics are collected and quickly converted into process improvements. Process Weaver® features include:

- An intuitive graphical process editor (definition and enactment),
- Control level integration with CASE, program management, and business applications,
- Import/export to and from program management tools,
- A client/server architecture, and
- Use on multiple platforms (e.g., Unix, AIX, Windows).

ATTENTION! In is strongly recommended you require your contractor to use Process Weaver® (or something equally effective) and that your program office has on-line access to the tool.

REQUIREMENTS AND DESIGN TOOLS

Requirements analysis and design tools (called Upper CASE tools) have functional capabilities in eight categories. These categories and their detailed functional characteristics are itemized on Table 10-10 (below).

NOTE: See STSC publication, *Requirements Engineering and Design*, October 1995, for evaluations of approved commercial requirements ad design tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

Requirements Engineering Environment (REE)

REE is a requirements engineering modeling and simulation environment that validates the critical aspects of software and systems prototypes. REE is composed of two tools: *Proto* and the *Rapid Prototyping System (RIP)* and an integration component which allows each tool to send/ receive data to/from one another. The REE is hosted on a Sun platform and runs under the Unix operating system.

Proto supports the prototyping of functional requirements by modeling and executing operational capabilities to check information completeness and correctness as it flows through the system. Proto creates a logical (software) model which captures functional software specifications and supports dataflow and object-based techniques for software representation. If the software is targeted for a parallel or distributed environment, a physical (hardware) model can also be created and associated with the software model. Proto addresses high-level architectural issues, such as *software-to-hardware* mappings and specific hardware architecture tradeoffs. It provides an interpreter facility for model execution which produces performance statistics after an interpretation session.

RIP is a suite of graphical tools for prototyping human-machine interfaces which model screen content and layout, as well as execute associated functions for validating user interface requirements. RIP uses a "*MacDraw-like*" (Apple Macintosh) editor to create prototype objects. Menus specify the activities to occur during prototype execution such as removing and displaying objects. RIP has a world database facility which graphically extracts geographical areas and incorporates them into the prototype. Prototypes can be developed with static or real-time dynamic displays. *[For about REE, see*

UPPER CASE CATEGORIES	FUNCTIONAL CHARACTERISTICS
Information Capture	• System functional descriptions • Data descriptions of system functional interfaces • Data descriptions of system input/output device interfaces • System logical behavior • System timing behavior • Hardware/software context • Software architecture structure • Software process definitions • Software data structures • Software process control • Software process concurrency • Software interprocess data communication • Software interprocess synchronization
Methodology Support	• Real-time structured development • Structured analysis • Structured design • Hatley/Pirbhai extensions • Ada-based object-oriented design • Entity relationship modeling • Petri nets • State charts • Axiomatic specification
Model Analysis	• Consistency checking • Completeness checking • Data normalization analysis • Man-machine interface analysis • Behavior analysis • Scenario-based analysis • Exhaustive model analysis
Requirements Tracing	• Extraction of requirements from system and software documentation • Inputs from electronically-scanned hard copy • Multiple requirements baselines • Tracing of system requirements to software requirements • Tracing of requirements to software design • Tracing of requirements to source code • Tracing of requirements to software and system test
Data Repository	• Data repository • Relational database type • Object database type • Support both text and graphics • Query capability • Access control capability • Concurrent access to entities • Contain program information • Contain requirements documents • Contain design specifications • Contain source code • Contain test descriptions and procedures • Capacity artificially limited • Support interactive cross-referencing configuration management capability

Table 10-10 Upper CASE Tool Functional Characteristics

Volume 2, Appendix A for information on how to contact Rome Laboratory.]

Sammi Development Kit (SDK)

Sammi is a prototyping tool that builds graphical user interfaces representing data, commands, and events of multiple processes which are connected to multiple data sources, either locally or across networks. It can be used for distributed C2 and other real-time systems development. It has been used on the **Canadian Automated Air Traffic System (CAATS)**, a gas equipment monitoring system, a British railway monitoring system, and a major US highway monitoring system.

Sammi consists graphics tools and a client/server architecture that separates the user interface from the back-end control system. This separation allows concurrent application with user interface development. It also provides an open environment for future upgrades or enhancements to the core application. The Sammi toolset consists of:

- ***Sammi Format Editor*** is a design tool for creating the interface using drawing primitives and *drag-and-drop* tools. The display building session results in a binary image of the window interpreted by its runtime environment.
- ***Sammi Runtime Environment*** is a group of processes that manage all interactions between the user interface, the application, the XWindow system, Motif, and Unix.
- ***Sammi Application Programming Interface (API)*** is a library of functions that enable communications between runtime environment(s), all end-user applications, and data sources. *[Sammi is developed by Kinesix, Houston, Texas.]*

AdaSAGE

AdaSAGE (developed by the Department of Energy) is a set of utilities to facilitate rapid construction of Ada systems. Its capabilities include database storage and retrieval (SQL™ compliant), graphics, communications, formatted windows, on-line help, sorting, and editing. It operates on MS-DOS, Unix System V, and OS/2 platforms. Its applications can run in the stand-alone mode or in a multi-user environment. AdaSAGE is not a development environment in the same sense as the Rational Environment,™ but uses Ada as a

UPPER CASE CATEGORIES	FUNCTIONAL CHARACTERISTICS
Documentation	• Support graphics/text integration • Completely compile a document • On-line templates • Automatic generation of documentation • Rapid draft hard copy • Interface to other document generators • Desktop publishing interface
Data Import/Export and Standardization	• Between tool kit components (intraoperability) • With other tools (interoperability) • CAIS-A interface standards/protocols supported • PCTE compliance • Compatibility with the evolving I-CASE environment
Reusability support	• Support for library design components

Table 10-10 Upper CASE Tool Functional Characteristics (cont.)

language with its own database and an SQL™ interface. It has the capability to easily embed informational databases into complex Ada applications. A developer using the AdaSAGE development system can design a product tailored to a specific requirement with quality performance and flexibility. Originally, AdaSAGE was a text-based display system, but the latest version changed the display format to a graphical database which produces charts as well as reports. Its *search-and-retrieve* mechanism is 10 times faster than other database-type applications in its class.

AdaSAGE is a standard development tool at the Marine Corps' Computer and Telecommunications Activity, Quantico, Virginia, where it is used to develop 55% of all applications. They have found it often outperforms commercial DBMS products and has reduced their development time by 50%. Beginning AdaSAGE video training materials are sufficient to train a high school graduate (with one month prior Ada training) to become an efficient Ada programmer in only 2 weeks. After 6 weeks of using the advanced training materials, the same programmer is as productive as an Ada programmer with 6 months experience. Marine programmers have rewritten many applications developed in the late 1980's adding new functionality by implementing AdaSAGE enhancements, while simultaneously reducing SLOC, increasing execution speed, reducing execution size, and improving maintainability.

The Marines and the Army are also using AdaSAGE for MISs they deploy with field units for applications such as unit inventories, personnel data, training records, and other personnel files. AdaSAGE allows this information to be uploaded into locally maintained databases

for personnel and inventory updates and ordering. AdaSAGE was used by the STSC on the PC toolbox and an **AdaSAGE User's Group** has been formed to assist product users. With their help, technical difficulties formerly wrestled with for days are now often be solved with a single phone call. [DePASQUALE93] *[This product is free (public domain). See Volume 2, Appendix A for information on how to contact the AJPO (discussed below).]*

AdaSAGE Success Story

According to the Ada Information Clearinghouse (AdaIC) (discussed below), on July 27, 1994, the **Type Commander Readiness Management System (TRMS)** won the category "*Best Object-Based Application Developed Using Non-Object Tools*" at the Object World Conference in San Francisco. Developed by the Naval Computer and Telecommunications Area Master Station LANT (NCTAMS LANT), the system defeated Pacific Bell, the other finalist, and 60 other contenders. The winning Ada team used the AdaSAGE development environment to re-engineer the legacy Type Commander Headquarters Automated Information System (THAIS), consisting of 2,000 applications and 2.8 million lines-of-COBOL. Their first program, they completed it on-time, within budget.

TRMS is the successful re-engineering of a legacy COBOL system into a state-of-the-art, object-based Ada application. A multifaceted, comprehensive information system serving diverse information needs of six Navy Type Commander staffs, it supports the business areas of combat readiness, casualty reporting, aviation maintenance, logistics, inspections and training. The re-engineering effort entailed information engineering, rapid prototyping, and enhancing the existing system and its documented backlog of change requests. TRMS, a distributed application, written in Ada 83, and running in a client server PC LAN environment, consists of over 150,000 lines-of-Ada. Of these, approximately 40% are calls into AdaSAGE, representing an estimated 150,000 additional lines-of-Ada. Using AdaSAGE allows developers to apply some object-oriented techniques to Ada 83. With internal reusable libraries representing approximately 9% of the application, the NCTAMS LANT contributed its library of reusable Ada components (over 12,000 lines-of-code) to the Navy reuse center. After being successfully fielded at six Navy Type Commanders, TRMS has exhibited the following advantages over the original COBOL THAIS system:

- The cost of equipping the six sites with server hardware equated to the cost of maintaining the THAIS system for just one year.
- The 2.8 million lines-of-COBOL have been reduced to just 245,000 lines-of-Ada.
- The station reduced its TRMS maintenance staff from 30 to 12 engineers.
- As a LAN-based system, TRMS is available to more staff members than with the THAIS system.
- As a PC-based product, TRMS can be used by any activity that has a PC or PC LAN.
- The usefulness of the system has expanded beyond its original customer base, thereby further leveraging the investment in its development.
- TRMS Ada components have been reused in other development programs. Reusing one major component can save approximately $50K in development cost.

The program has proven to be a cost effective alternative to the continued maintenance of the legacy system. The object-based nature of the Ada language, combined with the object-oriented approach of the Ada/AdaSAGE development environment have helped make software development easier and more cost effective. The benefits of the object-based/object-oriented approach continue to multiply as these methods are applied to future development programs. *[See Volume 2, Appendix A for a source of additional information on the TRMS and on how to contact the AdaIC.]*

Teamwork®/Ada

Teamwork®/Ada (developed by Cadre Technologies, Inc.) is an graphical modeling, object-oriented design and documentation tool, integral to I-CASE, which provides full notation and checking support for building, storing, reviewing, and maintaining complex Ada designs and code. It has an extended Buhr notation set for hierarchical navigation and the ability to graphically model an Ada design architecture. The *Source Builder* and *Design-Sensitive Editor (DSE)* features support iteration between design and code. *Source Builder* analyzes design diagrams and produces compilable Ada code frames for program unit bodies as well as specifications by checking diagrams against established criteria to verify design integrity. The *Teamwork/DSE* is a configurable, language-sensitive editor that enforces the graphical design specified by its *Ada Structure Graph Editor* feature. It also provides interface, multi-window editing, user-defined menus, custom key binding, user-defined macro routines,

and formatted templates. Teamwork®/Ada allows the addition of information to the code generated directly from the tool-specified, architectural design to assure consistency between the design and Ada source code. It has a reverse-engineering capability to help document and maintain existing code.

The *Ada Structure Graph (ASG)_Builder* feature reads existing Ada code and creates ASGs that reveal the existing source code's architecture. The DSE provides the functionality to propagate ASGs with actual source code. The combination of the ASG_Builder and DSE provides the capability to document, reuse, re-engineer, and maintain existing Ada code. Teamwork®/Ada contains a cross-referencing tool to locate object dependencies in any given module, to identify internal objects and where and which ones are being referenced. This helps in determining the objects complexity and architectural change ripple effects. The tool generates Tables of Contents (lists of figures, diagrams, and tables) and bullet/numbered lists and increases productivity and quality at each development stage. Teamwork®/Ada components include:

- Interactive graphical editors,
- Data flow diagrams, process specifications, data dictionaries, and entity relationship diagrams,
- A syntax-directed editing system (Windows),
- A program library,
- Model configuration management functions,
- SQL™ query and browsing functions,
- Cross-referencing,
- An annotation facility for note creation (notes may be associated with any Teamwork®/Ada object),
- Consistency checking for verifying requirements, design, and data dictionary elements, and
- A program data activity facility. Figure 10-7 (below) summarizes Teamwork®/Ada's capabilities.

Common Object Request Broker Architecture (CORBA)

CORBA is a commercially-available specification for an object-oriented messaging system. An implementation of this specification, called the *Object Request Broker (ORB)*, provides an interface between independently-developed, object-oriented applications. Acting as a go-between, the ORB is a communications mechanism by which objects make requests and receive responses.

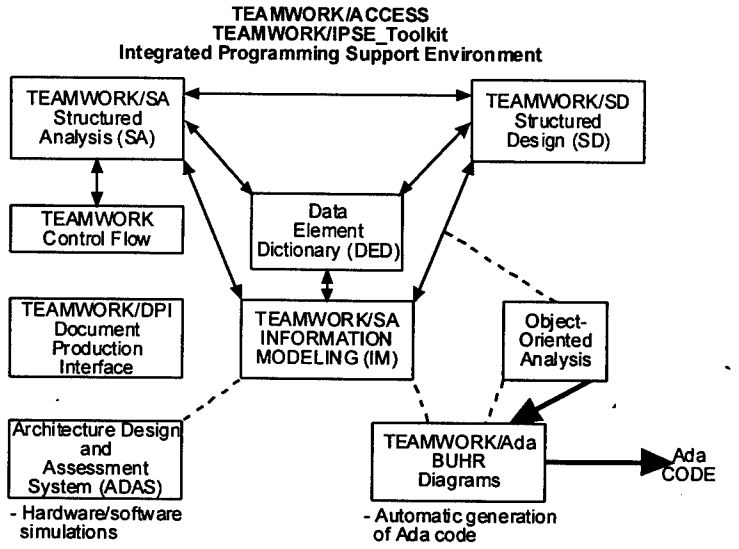


Figure 10-7 Teamwork®/Ada Overview

The CORBA specification was developed by a (300-member) industry consortium, the **Object Management Group (OMG)**. The OMG comprises many key industry software producers (large and small), such as Digital, Hewlett-Packard, IBM, Sun, and other software vendors. CORBA, part of a larger OMG specification, the *Object Management Architecture (OMA),* has four main components:

- ***Object services*** are lower-level object interfaces, such as persistence and access control.
- ***Common facilities*** provide higher-level standardized interfaces to common application services, such as mailers and printers.
- ***Application objects*** are the arbitrary collection of independently-developed applications (e.g., spreadsheets, word processors).
- ***Object request broker (ORB)*** provides the communication mechanism by which objects make requests and receive responses by locating, activating, or invoking the appropriate operations. Interfaces between the ORB and objects depend on, and make use of, an **interface definition language (IDL)**. These interfaces (called IDL specifications) provide a public API to developers.

A Sun product built using CORBA is called **Distributed Objects Everywhere (DOE)**. Digital's implementation (part of the COHESIONWorks product originally called *ACAS)* is now called *Object Broker*. Another implementation, called *Orbix* (marketed by

an Irish company, Iona Technologies) is available on a variety of platforms, including Sun. The OMG is incrementally developing specifications for the object services and common facilities. [CARNEY94]

TESTING TOOLS

For software to be truly reliable, it must undergo rigorous testing and verification throughout its development. Automated maintenance of test environments is an important productivity enhancer because programmers spend a substantial amount of time building and rebuilding them. Automated tools perform this function with greater efficiency and accuracy than manual methods — in less time, for less money. A well-organized test environment saves time during unit testing, systems testing, systems reviews, and quality control. **Computer-aided software testing (CAST)** tools, classified by the STSC, are presented on Table 10-11 (below). *[NOTE: Tools that manage test resources and support testing of requirements and designs are grouped under the management and requirements/design tool categories above.]*

NOTE: See STSC publication, *Software Test Technologies Report*, August 1995, for evaluations of approved commercial testing tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

Rate Monotonic Analysis for Real-Time Systems

Real-time systems are often seen as a "*niche*" within the software world — a crucial niche. Real-time software is often embedded in life-critical systems (e.g., avionics and other transportation systems, space stations, patient monitoring equipment, process control systems in chemical processing and nuclear power plants, and high-energy physics experiments). [FOWLER93] In these systems, multiple software tasks compete for limited resources [e.g., the central processing unit (CPU)]. Typically these tasks, such as monitoring altitude, monitoring cabin pressure, or controlling the fuel injection level on an aircraft, have different priorities, can occur at regular or erratic intervals, and require varying amounts of CPU resources to complete their jobs. Real-time systems also must complete critical tasks (e.g., the lowering of landing gear) within finite deadlines — or the entire system is at risk. Without the appropriate handling of

TEST TOOL CATEGORY	TEST TOOL FUNTIONALITY
Test Resource Management Tools	• Configuration managers • Program managers
Requirements and Design Test Support Tools	• Analyzers for software plans, requirements, and designs • System/prototype simulators • Requirements tracers • Requirements-based test case generators • Test planners
Implementation and Maintenance Test Support Tools	• Compilers • Source code static analyzers - Auditors - Complexity measurers - Cross referencing tools - Size measurers - Structure checkers - Syntax and semantics analyzers • Test preparation tools - Data extractors - Requirements-based test case generators - Test data generators - Test planners • Test execution tools (dynamic analyzers) - Assertion analyzers - Capture-replay tools - Coverage/frequency analyzers - Debuggers - Emulators - Network analyzers - Performance/flow/timing analyzers - Run-time error checkers - Simulators - Status displayer/session documenters - Test execution managers - Validation suites • Test evaluators - Comparators - Data reducers and analyzers - Defect change trackers

Table 10-11 STSC Test Tool Classifications

schedules and priorities, a lower priority task of relatively long duration (e.g., intermittent monitoring of passenger cabin pressure) can monopolize the CPU at the expense of a highly one. In real-time applications, system reliability depends, not only results, but on that point in time when outputs are generated. The quality attributes for real-time systems include:

- **Respondability**. Predictably fast response to urgent events.
- **Schedulability**. Schedulability is the degree of resource utilization at or below which timing requirements of can be assured (i.e., the number of timely transactions per second).
- **Stability**. Stability under transient overload is when the deadlines of selected critical tasks are guaranteed even though the system is overwhelmed by events and cannot meet all task deadlines. [SHA93]

Rate monotonic analysis (RMA) is a simple, practical, mathematically-sound method for guaranteeing all real-time timing requirements are met. RMA allows engineers to understand and predict the timing behavior of real-time software to a degree not previously possible. The **Rate Monotonic Analysis for Real-Time Systems (RMARTS)** program at the SEI demonstrated how to design, implement, troubleshoot, and maintain real-time systems using RMA to a degree not previously possible. *Rate monotonic scheduling theory* and its method of application, RMA, is a scientific approach that can be used before system integration to determine how well schedule requirements are met, and under what conditions task completion is guaranteed. RMA solves difficult problems early in development, and when properly applied, results in significant savings, not only of CPU resources, but of system development and operational resources, such as hardware. [FOWLER93]

Generalized Rate Monotonic Scheduling (GRMS) theory (as cited by the SEI) is a useful tool for meeting real-time requirements by managing system concurrency and timing constraints at the tasking and message passing levels. In essence, this theory assures all tasks meet their deadlines as long as system utilization lies below a certain threshold and appropriate scheduling algorithms are used. This method forces analytic, engineering discipline on real-time systems development and maintenance. [SHA93]

NOTE: See the article in Volume 2, Appendix P, Chapter 10 Addendum B, "*Rate Monotonic Analysis: Did You Fake It?*"

AdaQuest

AdaQuest is an Ada quality evaluation and testing toolset. Used during detailed design, coding, testing, and maintenance, it is a collection of static and dynamic analysis tools for assessing code quality, measuring test thoroughness and run-time performance, and capturing information needed to develop, maintain, reuse, and re-engineer large Ada systems. The *Static Analyzer* feature detects logic errors (e.g., infinite loops, unreachable statements, and data-flow anomalies), creates global cross references (e.g., identifies dependencies for objects, types, exceptions, and subprograms, or for user-defined symbols), illustrates software structure (e.g., application call graph, control-flow structure, & WITH clause dependencies), and identifies violations of common coding guidelines to improve program maintainability, portability, and reliability. The *Dynamic Analyzer* feature reports unit and branch execution coverage, measures performance timing (i.e., the minimum, maximum, and average time spent in user-specified portions of an application), and maintains test histories reflecting incremental results of a single test run or cumulative results over a set of test runs. The benefits of using a verification tool, such as AdaQuest, are summarized on Figure 10-8.

AdaQuest has automated Ada metrics collection and analysis capabilities that support the full spectrum of development activities. It allows engineers to incrementally measure progress and check compliances with application-specific quality standards, and provides program managers with visibility into the development process (i.e., SEI CMM[SM] Level 2 and above). AdaQuest features include:

- Rome Laboratory's *Software Quality Framework*,
- Cyclomatic complexity analysis,
- SEI software size measurements,
- SPC Ada Quality and Style Guidelines conformance checks, and
- Language feature profiles.

All metrics data can be exported to spreadsheets, databases, and popular public-domain graphics utilities for additional analysis. Built on the AJPO-adopted **Ada Semantic Interface Specification (ASIS)** *[discussed below]*, it has an open interface to the semantic content of an Ada library which makes it plug-compatible with any ASIS-supported Ada complier (e.g., Rational/Apex; Rational/Verdix, Alsys/RISCAda, and Alsys/AdaWorld). *[More information about AdaQuest can be obtained from the STSC.]*

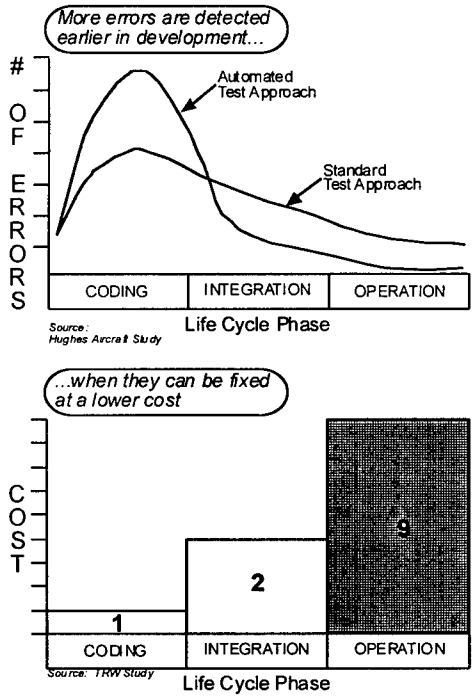


Figure 10-8 Benefits of Automated Code Verification

AdaWISE

AdaWise is a suite of "*light-weight*" verification tools. Development was co-sponsored by STARS and Rome Labs and implemented by Odyssey Research Associates (ORA). The AdaWise toolset reflects embedding of formal Ada verification concepts into tools that are usable and useful for all Ada programs. AdaWise supports validation of properties, including the absence of run-time anomalies, such as: incorrect order dependence, erroneous program executions, and accessing program variables before they have been assigned a value. This new technology transitions formal methods concepts into easily usable, lightweight tools. The AdaWise toolset utilizes the ASIS Ada Semantic Interface Specification and thus also serves to validate the utility of the ASIS STARS sponsored product. AdaWise has been integrated with Softbench as part of the Unisys STARS SEE. *[NOTE: "Light-weight," as used here, refers to the application of formal verification concepts to validation and testing problems rather than full-scale provable specifications.]*

AdaTEST

AdaTEST is used to test mission-critical Ada software (e.g., aircraft avionics systems, weapons systems, air traffic management systems, and nuclear reactor control systems). It combines support for dynamic testing (via a full test harness facility) with test coverage and static analysis (e.g., coding and complexity metrics) and can be executed in host and target environments. Its testing capabilities include statement, decision, condition, and exception coverage. Table 10-12 summarizes AdaTEST's features. *[See Volume 2, Appendix A for information on how to contact the tool vendor.]*

MathPack

MathPack is an Ada mathematical library with over 350 mathematical subprograms in 20 generic Ada packages. Subprograms provide solutions to a range of mathematical problems including linear systems, eigensystems, differential equations, integration, interpolation, transforms, special functions, elementary functions, basic linear algebra, random numbers, probability, and statistics. It also defines data types and numerical, physical, and chemical constants. MathPack's subprograms are based on proven numerical algorithms, such as LINKPACK for linear systems, EISPACK for eigensystems, and QUADPACK for integration. It also provides a binding to the Generic Package of Elementary Functions (GPEF). *[Contact the STSC for more information about MathPack.]*

McCabe Design Complexity Tool

The **McCabe Design Complexity Tool** calculates the McCable Design Complexity, analyzes a application's design, and automatically generates design paths that show the calling structures of modules. Also generates subtrees that graphically represent the calling structure of an application. It can be used to monitor the effect of system of changes and the insertion of new modules on existing modules. *[See Volume 2, Appendix A for information on how to contact the vendor of this tool.]*

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on McCabe Complexity Analysis. Also see the article in Volume 2, Appendix P, Chapter 8 Addendum B, "*Software Complexity*," by Thomas McCabe.

AdaTEST FEATURES	FUNCTION
Dynamic Testing	• Software execution • Data checking • Exception checking • Test script generation • Simulation • Timing facilities • Check occurrence of physical events
Coverage Analysis	• Code branch coverage • Statement coverage • Exception coverage • MCDC coverage • Boolean operator/operand coverage
Static Analysis	• File/unit source code statements • File/unit declaration statements • Number of operators (of each type) • McCabe's cyclomatic complexity metric • Halstead metric • Harrison metric • Average and maximum nesting level • Units in source file • "*Use*" clauses (e.g., LOOP statements, IF statements, etc.)
AdaTEST Script Generation (ATS) Module	• Isolation testing scripts • Missing Ada unit (stubs) interaction simulation scripts • Test case definition (TCD) scripts • Coverage scripts • Static analysis facilities scripts
AdaTEST Analysis (ATA) Module	• Static metrics • Data coverage analysis • Path coverage • Warning level checks • Dynamic analysis • Assertion checks • Package-based reporting • Average and total values • Flowgraphs
AdaTEST Harness (ATH) Module	• Generic array checks • Self-protection from corrupt test code • Check observation expected result • Stub matching • Batch file processing • Exception failure count • Full on/off directive

Table 10-12 AdaTEST Features and Functions

McCabe Instrumentation Tool

The **McCabe Instrumentation Tool** is a dynamic testing analysis tool that graphically and textually represents tested and untested paths. It cumulatively monitors code execution and generates all remaining unit-level and integration tests needed. The tool can also be used to isolate error-prone subtrees and modules. *[See Volume 2, Appendix A for information on how to contact the tool vendor.]*

McCabe Slice Tool

The **McCabe Slice Tool** is a data-driven visualization tool that traces data through the system's architecture. It aids in relating the software's functional execution to its internal structure. It highlights lines-of-code and the graphical section, *"slice,"* of a system's architecture that was traversed during the transaction. The tool can be used for debugging, downsizing, requirements traceability, and for identifying redundant and reusable code.

Analysis of Complexity Tool (ACT)

The **Analysis of Complexity Tool (ACT)** examines the control structure of a module. It calculates McCabe Cyclomatic Complexity, produces module flow graphs that graphically represent control structure, and generates a basic set of test paths and corresponding test conditions. *[See Volume 2, Appendix A for information on how to contact the vendor of this tool.]*

Battlemap Analysis Tool (BAT)

The **Battlemap Analysis Tool (BAT)** analyzes source code at the system level and calculates McCabe essential complexity. The primary applications of BAT are in software maintenance, reverse engineering, systems analysis, and program management. Structure charts, called *Battlemaps,* graphically represent system design. Modules are identified as testable, maintainable, or unmaintainable. The tool identifies error-prone modules and subsystems and user-defined indicators that reflect work completion, code acceptance, or the presence of design specifications. It calculates McCabe Design Complexity, analyzes an application's design, and automatically generates design paths that show module calling structures. Subtrees, graphically representing the calling structure are also generated. The tool can be used to monitor the effect that module changes and new module insertions of have on a system. It provides a basic set of

subtrees necessary to perform rigorous integration testing, and by showing the architectural design, it aids in reverse engineering. *[See Volume 2, Appendix A for information on how to contact the tool vendor.]*

TestMate

TestMate (developed by Rational Software Corporation) is a software analysis tool that tests life- or safety-critical systems. Based on technology developed to test **Boeing 777** avionics source code (two million lines-of-Ada), TestMate meets FAA requirements for **modified condition decision coverage (MCDC)**. MCDC ensures code is fully tested by checking software variables (i.e., conditional expressions). Whenever a statement can have more than one outcome (e.g., an *if-then* formulation), each outcome and conditional expression of that outcome must be tested. Because no unnecessary software is allowed in a flight box, TestMate's checking procedure ensures no unintended functionality shows up in an avionics system while in flight.

Also as part of the 777 program, Boeing worked with FAA engineers in developing the TestMate algorithm that identifies untested code. Prior to its development such testing was performed manually. TestMate's capabilities include integrated test management, test-results analysis, and noninvasive coverage analysis. Its test management features provide a consistent framework to manage creation, storage, and retrieval of test cases and to determine every potential outcome of each conditional expression. It automates the testing process by combining test cases into test lists and executing those lists against multiple testing scenarios and by defining specific conditions, parameters, and scenarios for each test case.. The test management framework supports reuse of test cases across different phases of the software testing process. The TestMate toolset is being used by the Air Force, the Naval Research Laboratory, Westinghouse Corporation, Honeywell, and other avionics vendors working on DoD programs. *[See Volume 2, Appendix A for information on how to contact the tool vendor.]*

NOTE: See STSC publication, *Documentation Technology Report*, April 1995, for evaluations of approved commercial documentation tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

RE-ENGINEERING TOOLS

The STSC has identified several re-engineering tools domains, as illustrated on Figure 10-9.

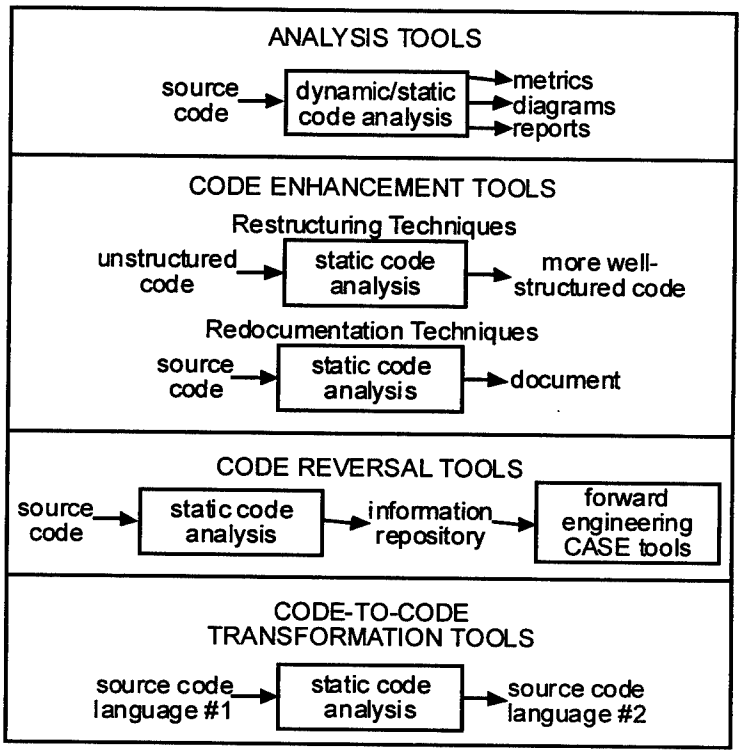


Figure 10-9 Re-engineering Process Models [STEVENS92]

NOTE: See Chapter 11, *Software Support*, for a discussion on re-engineering.

- **Analysis tools**. These tools depict the attributes of a software application through dynamic or static code analysis and produce metrics, diagrams, and reports. The source code remains unaltered through this process. Process outputs include complexity analyses, reusability analyses, and variable cross-references.
- **Code enhancement tools**. These tools include restructuring and redocumentation tools. Restructuring tools automate source code restructuring and produce code which is functionally equivalent

to its input. Redocumentation tools produce documentation through analysis of source code.

- **Code reversal tools**. These tools automate reverse engineering by reading and abstracting source code. They create a repository of information compatible with most CASE tools.
- **Code-to-code transformation tools**. These tools automate the translation of source code from one language to another producing a functionally equivalent output. [STEVENS92]
- **Data name rationalization tools**. These tools enforce uniform naming of the same logical data item across all software products.
- **Data re-engineering tools**. These tools perform all the re-engineering functions associated with source code (i.e., reverse engineering, forward engineering, translation, redocumentation, restructuring/normalization, and retargeting) which act on data files.
- **Reformatting tools**. Redocumentation tools which make source code indentation, bolding, capitalization, etc., consistent.

NOTE: See STSC publication, *Re-engineering, Volume 1*, 1995/ 1996, for evaluations of approved commercial re-engineering tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

SORTS

The **SOftware Reliability Modeling and Analysis Toolset (SORTS)** was designed to remedy problem software delivered with latent development errors (low reliability) and poorly configured and documented design (low maintainability). It was developed by Science Applications International Corporation (SAIC) to support the **Defense Meteorological Satellite Program (DMSP)**, the **Global Positioning System (GPS)** program, and the **Air Force Satellite Control Network Common User Element (AFSCN/ CUE)** program. Developed in conjunction with hardware reliability and maintainability (R&M) modeling and analysis tools, SORTS has the capabilities to:

- Measure software R&M,
- Generate software metrics,
- Provide the basis for correlating metrics to R&M,
- Assess software quality,
- Improve software development and support process maturity levels,
- Track growth and decay trends of R&M parameters,

- Reduce time spent on change request analysis, and
- Allow realistic measures for maintenance manhours.

SORTS was also designed to support measurement programs through its ability to:

- Process, extract, and use software size, quality, effort, and schedule metrics to improve the software support process,
- Indicate which metrics are predictors of software R&M and life cycle costs,
- Track and evaluate progress in software support process improvement, and
- Assess the quality of the software product. *[For information on how to contact the SORTS vendor, SAIC, see Volume 2, Appendix A.]*

MEASUREMENT TOOLS

For processing metrics data, tools are quite mature and should be given priority over manual, time consuming techniques. Many tools exist for collecting and analyzing metrics data, ranging from error checking, software consistency checking, and error analysis to complexity measurement. When using analysis tools, you should compare metrics taken before and after the tools were employed. [YOURDON92] Tools such as **AdaQuest**, an automated defect detection and verification tool *[discussed above]*, and **AdaMAT**, an automated software quality verification tool, are examples of tools you might consider for your metrics program. These tools are largely automatic and nonintrusive, and can reduce manual efforts associated with metrics collection by 80% to 90%.

NOTE: Many of the testing tools discussed above have measurement data collection and reporting capabilities.

AdaMAT

AdaMAT (developed by Dynamics Research Corporation and marketed by Rational) is a comprehensive tool for static analysis and quality metrics that helps in understanding Ada code and in locating anomalies that cause maintenance, reliability, and portability problems. AdaMAT checks for adherence to hundreds of Ada quality principles, the use of which improve software quality. Other metrics address

specific programming concerns, such as code simplicity, modularity, self-descriptiveness, exactness, clarity, and independence. It produces management reports and reduces the effort required to perform effective design and code walkthroughs by automatically identifying unnecessarily defect-prone, machine-dependent, or complex software characteristics. AdaMAT has life cycle applicability by generating reports that reveal critical code details at specified development stages, making it easier to identify problems and errors early. Tool output can also be displayed in graphic format on an IBM PC or compatible, via AdaMAT's **Metrics Display Tool**. The benefits of using AdaMAT as a measurement tool are:

- Detection of downward quality trends resulting from failure to adhere to software engineering standards. Manpower and computing resources can be redirected before poor quality code causes schedule slips and cost overruns.
- Developer's skills and computing resources can be allocated to tasks based on module characteristics (e.g., large amounts of tasking, numerical processing, specialized I/O, or machine dependencies).
- Productivity levels can be restored to acceptable levels after the intervention of testing or demonstration milestones.
- Consistent design and coding among different development groups and individuals can be assured.
- Individual programmer training requirements can be determined to improve their understanding and correct use of specific Ada features.
- Training programs can focus on specific quality principles to enforce their use. *[For more information, see Volume 2, Appendix A.]*

Amadeus Measurement System

The **Amadeus Measurement System** (by Amadeus Software Research, Inc.) is an automated software metric collection, analysis, reporting, graphing, and prediction system for software process and product analysis and improvement. Amadeus increases program visibility and facilitates informed technical and management decision making — enabling better, faster, and cheaper products. Amadeus provides a flexible system for collecting different types of metric data, ranging from automated data collection to interactive data entry. Example categories of metrics collected are: errors, changes, size, structure, cost, schedule, effort, and cycle time. Amadeus has a user-tailorable, user-extensible open architecture that can integrate Amadeus-supplied tools and foreign tools.

Amadeus provides a template language, similar to a macro language, which allows users to define and customize their own metrics. Users can define a template set that supports a particular metric set, and then share that set throughout an organization to facilitate consistent data collection and interpretation. Example template sets have been developed for SEI CMM[SM] Levels 2 and 3, Air Force Technical Performance Measurement (TPM) metrics, telecommunications In-Process Quality Metrics (IPQM), and numerous source code metrics including size and cyclomatic complexity.

The *Amadeus Basic Package* provides infrastructure, tools, and pre-defined templates for metric graphs, reports, data export, data import, and data entry. The *Amadeus Customization Package* provides full metric templates user-customization. The *Amadeus Prediction Package* provides metric prediction capabilities using classification analysis. The Amadeus source code metric collection tools analyze various source code languages (e.g., Ada, C, C++, Unix Csh scripts).

Amadeus generates PostScript files for metric graphs and ASCII files for metric reports. Metric data can be imported from an ASCII file or the "*tab-separated-value*" spreadsheet format. Metric data can be exported to an ASCII file or the "*comma-separated-value*" or "*tab-separated-value*" spreadsheet formats. Amadeus provides an application programmer interface for C or Unix Csh scripts that can be used for automatic data collection and integration with other tools. Amadeus metric templates, data formats, and application programmer interface are platform independent and can be reused without change across all Amadeus-supported platforms. Amadeus clients and servers are fully interoperable across all Amadeus-supported platforms, i.e., an Amadeus client can execute on one platform type (such as Sun, IBM, etc.) and send/receive data to/from an Amadeus server executing on another platform type (such as HP, Digital, etc.). The Amadeus products are available on these platforms: Sun Sparc SunOS 4.1.X, Sun Sparc Solaris 2.X, IBM RS/6000 AIX 3.2, DEC Alpha Digital Unix 3.X, DEC VAX VMS 5.5-2, HP 9000/700 HP-UX 9.0, and SGI Indy IRIX 5.3. *[For information on how to contact the Amadeus vendor, see Volume 2, Appendix A.]*

CONFIGURATION MANAGEMENT TOOLS

Configuration management (CM) for a large development effort, involving multiple software releases and developers, must be supported by automated tools. A CM toolset should not simply maintain files — it should support:

- Release management;
- Problem report management;
- Control of hardware, software, data, and builds;
- Status accounting;
- Audit trails; and
- The relationships among software products.

A highly-automated CM toolset is required when vast amounts of CM data are generated during large, software-intensive developments. Such a toolset should provide:

- A mechanism for storing, maintaining, and retrieving source and object code and its supporting documentation;
- A mechanism for automatically identifying all elements of a configuration item;
- A mechanism for interfacing the CM tool with the tools that build the deliverable software; and
- A mechanism to maintain a historical log of configuration changes by storing the incremental changes to each version so an old version can be recreated.

Configuration management tools can have all, or combinations of, the following features:

- Version control,
- Configuration support,
- Process support,
- Change control,
- Team support,
- Library/repository support,
- Security/protection,
- Reporting/query,
- Tool integration,
- Build support,
- Release management,
- Customization support, and
- Graphical user interface (GUI).

NOTE: See STSC publication, *Software Configuration Management*, September 1995, for evaluations of approved commercial configuration management tools. *[See Volume 2, Appendix A for information on how to contact the STSC.]*

Process Configuration Management Software (PCMS)

The **Process Configuration Management Software (PCMS)** (developed by Lockheed Martin Aeronautical Systems) is an example of a CM tool used on the F-22 Program. It allows software engineers from over 40 different vendors to work concurrently by automatically identifying problems within the mainstream program or with changes proposed by the independent vendor groups. The tool is used after the development process is properly planned and partitioned into separate work functions. F-22 software teams are working in parallel by using the tool which alerts them when independent inputs cause conflicts. As conflicts do occur, PCMS aids in understanding the relationships among the effected parts of the program.

On highly complex programs like the F-22, PCMS' configuration management function streamlines the approval process by providing a detailed audit trail, and by documenting who is making decisions, what they are, and when. Because decision-makers can log-on using their password on any terminal, personal accountability is strengthened. A fully-documented process control mechanism allows users to choose which tool features are loaded for their specific needs. No programming is required to adjust the model, which can be customized using pull-down menus.

PCMS can tightly control the software development process so no new code is written without a change authorization document. The user can also loosened the tool's control to encourage innovation and adjust the model's built-in 80% accuracy to suit individual accuracy needs. The F-22 software avionics change process is being managed though on-line change control meetings (traditionally involving travel and 3-4 hours of each team member's time). These meeting are now accomplished in about 5 minutes at team members' own locations. [NORDWALL95]

TECHNOLOGY SUPPORT PROGRAMS

There are several programs that provide assistance in transitioning technology from and to DoD programs. They include the:

- Ada Joint Program Office (AJPO),
- Advanced Computer Technology (ACT) Program,
- Computer Resource Technology Transition (CRTT) Program, and
- Embedded Computer Resources Support Improvement Program (ESIP).

Ada Joint Program Office

The **Ada Joint Program Office (AJPO)** was responsible for the original definition of Ada 83 and **Ada 95**. The AJPO also has responsibility for the validation/evaluation of Ada compilers, sponsorship of the **Ada Information Clearinghouse (AdaIC)**, the common Ada programming support environment (APSE) interface set, and the **Ada Technology Insertion Program**. In the **Defense Management Report Decision 918,** the AJPO was one of the activities recommended for transfer to **DISA**. This transfer to DISA's **Joint Interoperability and Engineering Organization** was one of many initiatives for consolidating DoD software standards activities under one agency. **Ada Technology Insertion Program** products have included:

- AdaSAGE enhancements,
- A Broad-Based Environment for Test (ABBET),
- Computer-aided Prototyping System for Real-Time Software,
- Ada Reuse in a Trusted Message Processing System, and
- Reusable Ada Products for Information Systems Development (RAPID).

AJPO bindings programs have included:

- Decimal arithmetic,
- GOSIP,
- POSIX (real-time),
- SQL,™
- XWindows, and
- 1553 data bus.

The AJPO provides a source of electronic information on the Ada language and Ada activities. The **Ada Information Directory** on the AJPO host contains all the files available on the **AdaIC Bulletin Board**. The **Ada 95 Directory** contains all the files on the Ada 95 bulletin boards and the tools directory contains pre-canned searches from the AdaIC products and tools database. The AJPO host is accessible to authorized users of the Defense Data Network (DDN) and other Internet networks. [AdaIC92] *[See Volume 2, Appendices A and B for information on how to contact the AJPO.]*

Ada Information Clearinghouse (AdaIC)

The **Ada Information Clearinghouse (AdaIC)** (sponsored by the AJPO) provides a full spectrum of information on the Ada programming language. Through flyers, databases, and a newsletter, information is available on Ada community events, working groups, research, publications, and issues. This information can be obtained in hardcopy or electronically from the bulletin board or the AJPO directory on the Internet. *[See Volume 2, Appendices A and B for information on how to contact the AdaIC.]*

PAL

The **Public Ada Library (PAL)** is a collection of Ada software, courseware, and documentation on the Internet. The purposes of PAL are to make Ada-oriented software, courseware, and documentation, released for public distribution (as shareware, freeware, GNU Copyleft, etc.), readily available to the public. It provides a convenient way for the Ada user community to exchange materials and ideas. It contains:

- A collection maintained by the Ada Information Clearinghouse (AdaIC),
- The **Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)** *[described in Chapter 14, Managing Software Development]*
- A repository at the University of Cincinnati maintained by the Department of Electrical and Computer Engineering, and
- A PAL-maintained collection containing the PAL Catalog, PAL LOTUS-123, dBase IV compatible database files, and other forms of catalog information.

Many organizations have developed cooperative relationships with the PAL contributing time, effort, user support services, and artifacts —either directly or indirectly. These organizations include: AdaNET,

the AJPO, the AdaIC Asset Source for Software Engineering Technology (ASSET) *[discussed in Chapter 4, Systems and Software Engineering]*, the Comprehensive Approach for Reusable Defense Software (CARDS) [discussed in Chapter *9, Reuse]*, Conservatoire National des Arts et Metiers (CNAM), Paris, France, the SEI, the Software Reuse Repository at the University of Maine, and the VHDL Repository at the University of Cincinnati. *[See Volume 2, Appendix B for information on how to access PAL on-line through the AdaIC.]*

ASIS

The **Ada Semantic Interface Specification (ASIS)** (sponsored by AJPO) provides an interface between an Ada library and the tools within a software engineering environment. ASIS is an open, callable interface giving CASE tool and application developers access, through predefined formats, to the semantic information contained in any Ada compiler. It can increase the number of tools available to the developer by making them more easily portable to competing compiler environments. Independent of underlying compiler library implementations, ASIS supports portability of CASE tools while relieving ASIS users from having to understand the complexities of internal data representation in an Ada compiler library. For example, a metrics tool can use ASIS to evaluate Ada application code. The symbolic name, type, and usage of each object in the symbol table can be obtained through the ASIS interface to support metrics tool requirements. Client/server communications between Ada programs occurring on separate processors can also benefit from ASIS. For instance, ASIS is currently being used to analyze messages transmitted from a satellite to a ground station. This technology can have significant cost saving benefits for any application doing data reduction for post mission analysis. Examples of tools that can benefit from ASIS are:

- Symbolic debuggers,
- Test tools,
- Design tools,
- Reverse engineering and re-engineering tools,
- Metrics tools,
- Style checkers,
- Correctness verifiers, and
- Automatic document generators. *[See Volume 2, Appendix A for information on how to contact the AJPO about ASIS.]*

Advanced Computer Technology (ACT) Program

The **ACT** program [based at Rome Laboratory *(discussed below)*], is a basic research and development program that develops and demonstrates technologies to control cost, reduce risk, and increase the efficiency of embedded software and technologies. Its objective is to take recent software technology advances in artificial intelligence, distributed databases, networking, and open systems architectures to a specified prototype stage. These efforts are focused on improving the survivability of distributed C2 systems and the effectiveness and performance of weapon systems.

Computer Resource Technology Transition (CRTT) Program

The **CRTT** program (Electronic Systems Center (ESC), Hanscom AFB, Massachusetts) addresses the problems of acquiring, developing, and supporting emerging C2 resources to reduce costs and improve the software development and support processes. CRTT is an engineering development program that transitions software technology development from laboratories, industry, and academia to operational users and SPOs. CRTT has a three-fold approach for accomplishing its mission:

- To establish the fundamental elements of an effective software technology transition methodology;
- To implement reusable software technology in the C2 domain; and
- To address the technologies/processes inherent in operating and maintaining a central repository of software, software algorithms, and reusable technologies that collectively encompass the development life cycle.

Embedded Computer Resources Support Improvement Program (ESIP)

The **ESIP** program (Ogden Air Logistics Center, Hill AFB, Utah) supports an infrastructure that transitions technology to improve the ability of software practitioners to meet weapon systems PDSS requirements. ESIP's goal is to transition technology from developers to customers who will then institutionalize it in their programs. This program provides for engineering services, hardware procurement, research and development, requirements definition, design, development, installation, testing, selective prototyping, and/or limited

COTS procurement. ESIP provides software productivity tools, a test bed for software technology insertion, automated software configuration management tools, and rapid prototyping to address changing threat scenarios and improvements in weapon systems effectiveness and performance. This is accomplished by increasing awareness through information exchange, aiding in software process improvement, and facilitating software technology transition.

TECHNOLOGY SUPPORT CENTERS

The organizations responsible for managing and evaluating tools for use on major DoD software developments include:

- Software Technology Support Center (STSC),
- Standard Systems Center (SSC),
- Software Engineering Institute (SEI),
- Rome Laboratory, and
- Oregon Graduate Institute Formal Methods Research.

Other sources for tools and evaluations are the Software Productivity Consortium *[discussed in Chapter 16, Managing Process Improvement]* and the Software Technology for Adaptable, Reliable Systems (STARS) program *[discussed in Chapter 9, Reuse.]*

Software Technology Support Center (STSC)

Software Technology Support Center's (STSC) (Hill Air Force Base, Utah) services are valuable when choosing an Ada support environment as they evaluate and compare a range of Ada support tools from the public and private sectors. The STSC acts as the *Consumer's Report* on software engineering technologies. Their mission is to:

> *improve software quality, productivity, and interoperability by promoting improved business practices, processes, and technologies. This includes increasing an organization's contact, awareness, understanding, usage, and adoption of software practices, processes, and technologies.* [PETERSEN92]

The STSC's primary areas of expertise are weapons systems and embedded C2 systems, but increasingly these evaluations are equally applicable to MIS. Their publications and monthly newsletter,

CrossTalk, are valuable in keeping your up-to-date on tends and current schools of thought on software engineering throughout DoD and industry.

NOTE: *CrossTalk*, distributed without charge to DoD and DoD contractor personnel, is highly recommended reading for its currency, professionalism, and technical content. *[See Volume 2, Appendix A for information on how to contact the STSC and be placed on CrossTalk's mailing list.]*

Standard Systems Center (SSC)

The **Standard Systems Center (SSC)** (Gunter Annex, Maxwell AFB, Alabama) was established to be a model software development, maintenance, and procurement organization for standard systems has become a software *"center of excellence."* Their primary areas of expertise are standard systems, MISs, and non-embedded C2 systems. The SSC mission is to increase user satisfaction through better processes for prioritizing requirements and increasing user participation and support. It aids software quality improvement by ensuring that reliable, maintainable software fulfills user mission requirements. The SSC bimonthly newsletter is a valuable source of information about MIS and non-embedded C2 software development. *[See Volume 2, Appendix A for information on how to contract the SSC about their products and services.]*

Software Engineering Institute (SEI)

The mission of the **Software Engineering Institute (SEI)** is to bring the ablest professional minds and the most effective technology to bear on software-intensive systems improvement. Its goal is to accelerate the practice of modern software engineering discipline throughout the defense community by establishing *standards of excellence*. The SEI, a federally-funded research and development center (FFRDC) sponsored by DoD, has the following characteristics:

- It is affiliated with a university (Carnegie Mellon University, Pittsburgh, Pennsylvania) granting doctoral degrees in software engineering and computer science;
- It has wide access to industry, academic, and US government data concerning software engineering technology;
- It is a nonprofit corporation outside the control of any profit-seeking concern;

- It has no proprietary interest in the production of software products, equipment, systems, or services for sale or profit;
- It is strictly prohibited from competing for business; and
- It retains a Board of Visitors composed of distinguished technical leaders from industry, academia, and Government to review its plans and accomplishments.

In pursuit of its mission, the SEI conducts specific programs and efforts in the areas of technology transition, research, education, and in support of DoD components. The **Advanced Research Programs Agency (ARPA)** has executive oversight of the SEI. HQ ESC acts as the SEI's administrative and contracting agent and maintains a Joint Program Office with it. *[See Volume 2, Appendix A for information on how to contact the SEI.]*

Rome Laboratory

The **Rome Laboratory** (Griffiss AFB, New York) has a long history of developing and applying software engineering technology. It also has a 40-year history in applying new software concepts to improve the automation of C3I functions. Rome has pioneered, or has expertise in, the following technologies:

- Software/system engineering environment frameworks,
- Software/system requirements engineering processes and workstations,
- Software product quality prediction and measurement *[NOTE: A consortium of aerospace companies is using and evaluating this technology],*
- Software engineering for distributed/parallel systems,
- Toolsets for developing and supporting mission planning and scheduling systems,
- The engineering/integration of artificial intelligence systems,
- "*Intelligent*" software engineering tools that understand both software engineering and application areas,
- Distributed, real-time computing and databases,
- Computer security including tools to specify and verify both policy and systems software, and
- 3-D and virtual reality interfaces for C3I systems.

In addition, Rome develops and/or fields high-tech, revolutionary system components or whole systems for:

- Signal intelligence,
- Intelligence data handling (IDHS),
- C2 decision aids, and
- Distributed C2 systems. *[See Volume 2, Appendix A for information on how to contact Rome Labs.]*

Oregon Graduate Institute Formal Methods Research

The **Oregon Graduate Institute Formal Methods Research** program specializes in formal methods for software development based on a belief that more reliable, better understood software can be produced by applying rigorous, mathematically-based reasoning to the software problem analysis, specification, and design process. Formal design methods are compatible with various technologies for structuring software, such as object-oriented design, rapid prototyping, implementation language choice, and software metrics. The goals of its formal research methods are:

- To obtain authoritative and complete specifications for software components and systems functional requirements;
- To verify selected properties of a software specification with the aid of automated tools;
- To capture design decisions taken at any level of abstraction;
- To automate the process of specification implementation;
- To improve the modularity and reusability of software designs; and
- To manage designs, verifications, and implementations throughout the software life cycle.

These techniques have the potential to lead to new methods for software design and life cycle maintenance, in which:

- Software components are designed by solution schemes specification;
- Components are integrated by algebra composition (abstract data types);
- Critical properties are verified by program specification proofs;
- Software is automatically generated from solution schema specifications, and programs are improved automatically by type-parametric program transformations; and

- All modifications to software are made at the specification level. *[See Volume 2, Appendix A for information on how to contact the Oregon Graduate Institute.]*

ADDRESSING TOOLS IN THE RFP

High-performance teams are more productive and develop higher quality software when they employ a properly automated toolset known as a **software engineering environment (SEE)**. To develop a successful software system, the offeror must have a comprehensive software production capability in place. The offeror's SEE must include tools, methods, management practices, and organizational resources. Their proposals must also describe a successful integration of software methodologies and tools. Included in these tools should be a process control mechanism (e.g., the use of **Process Weaver®** or equivalent) and a discussion on their prior experience in using that control mechanism and the software development and configuration management portions of the SEE on past programs of similar size and complexity, within the same domain.

Consideration of the ASC/SEE, or equivalent SEE, should be encouraged for weapon system software. Ideally, each subcontractor should employ an identical SEE. In fact, in all major software development acquisitions, regardless of domain, contractors should be encouraged to provide a common SEE for all subcontractors. Where the SEE and associated methodology are new for some subcontractors, the prime should provide risk mitigation considerations, including a robust education and training program for personnel of those organizations with no prior experience in using the SEE.

REFERENCES

[AdaIC92] AdaIC staff writers, "Accessing Ada Information Clearinghouse Electronic Services," *CrossTalk*, April/May 1992

[AGGARWAL95] Aggarwal, Rajesh and Jong-Sung Lee, "Case and TQM for Flexible Systems," *Information Systems Management*, Fall 1995

[ALLEN92] Allen, Woody, as quoted by the staff, "Snapshots from STSC-92," *CrossTalk*, April/May 1992

[CARNEY94] Carney, David, "Brief Description of the Common Object Request Broker Architecture," Software Engineering Institute, August 1994

[CRAFTS93] Crafts, Ralph E., "Megaprogramming in Practice — TRW's UNAS," Ada Strategies, Software Strategies and Tactics, Inc., Vol. 7, No. 6, June 1993

[DePASQUALES93] DePasquale, MAJ Gerald A., "Marine Corps AdaSAGE Experiences, "briefing, Marine Corps Computer and Telecommunications Activity, Quantico, Virginia, April 1993

[FOWLER93] Fowler, Priscilla, and Linda Levine, *Technology Transition Push: A Case Study of Rate Monotonic Analysis (Part 1)*, CMU/SEI-93-TR-29, ESC-TR-93-203, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1993

[GIBBS94] Gibbs, W. Wayt, "Software's Chronic Crisis," *Scientific American*, September 1994

[HART24] Hart, Capt Sir Basil Liddell, as quoted in Thoughts on War, 1944

[HUMPHREY90] Humphrey, Watts S., Managing the Software Process, Software Engineering Institute, Addison-Wesley Publishing Company, 1990

[HUMPHREY95] Humphrey, Watts S., personal communication to Capt Stanko (USAF), SAF/AQKS, September 15, 1995

[KENNEDY93] Kennedy, Paul, Preparing for the Twenty-First Century, Random House, New York, 1993

[MAHON08] Mahon, RADM Alfred, Naval Administration and Warfare, Little, Brown, & Co., Boston, 1908

[MARCINIAK90] Marciniak, John J., and Donald J. Reifer, Software Acquisition Management: Managing the Acquisition of Custom Software Systems, John Wiley and Sons, New York, 1990

[MARSHALL80] Marshall, BGEN S.L.A., The Soldier's Load and the Mobility of a Nation, The Marine Corps Association, Quantico, Virginia, 1980

[MOSLEY95] Mosley, Vicky, "Improving Your Process for the Evaluation and Selection of Tools/Environments," briefing presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[MOSEMANN92] Mosemann, Lloyd K., II, "Software Management," keynote closing address, Fourth Annual Software Technology Conference, Salt Lake City, Utah, April 16, 1992

[NORDWALL95] Nordwall, Bruce D., "Automated Tool Speeds F-22 Software Work," *Aviation Week & Space Technology*, October 30, 1995

[O'BERRY93] O'Berry, Lt Gen Carl G., as quoted in *Ada Information Clearinghouse Newsletter*, Vol. XI, No. 2, August 1993

[PETERSEN92] Petersen, Gary, "The Spectrum of STSC Support, *CrossTalk*, March 1992

[QUANN93] Quann, Eileen, "Software Will Change the Way We Live," Fastrak Training, Inc., Columbia, Maryland, September 1993

[RAJA85] Raja, M.K., "Software Program Management and Cost Control," *Journal of Systems Management*, October 1985

[RANDALL95] Randall, Richard, and William Ett, "Using Process to Integrate Software Engineering Environments," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[REICH91] Reich, Robert, *New Perspectives Quarterly*, Fall 1991

[ROYCE91] Royce, W.E., "Ada Technology Evolution, CCPDS-R: An Ada Success Story," TRW briefing, March 21, 1991

[ROYCE93] Royce, W.E., as quoted in "Megaprogramming in Practice — TRW's UNAS," Ralph E. Crafts, editor, *Ada Strategies*, Vol. 7, No. 6, June 1993

[SHA93] Sha, Lui, and Shirish S. Sathaye, *Distributed Real-Time System Design: Theoretical Concepts and Applications,* CMU/SEI-93-TR-2, ESC-TR-93-179, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1993

[SHATZ96] Shatz, Daryll, "Reducing Software Development Risk," Digital Equipment Corporation, 1996

[STEVENS92] Stevens, Barry J., "Linking Software Re-engineering and Reuse: An Economic Motivation," *CrossTalk*, August 1992

[TOBIN90] Tobin, L.M., "The New Quality Landscape: Total Quality Management," *Journal of Systems Management*, November 1990

[UTZ92] Utz, Walter J., Jr., Software Technology Transitions: Making the Transition to Software Engineering, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[YOURDON92] Yourdon, Edward N., Decline and Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992

[ZRAKET92] Zraket, Charles E., "Software Productivity Puzzles, Policy Changes," John A. Alic, ed., Beyond Spin-off: Military and Commercial Technologies in a Changing World, Harvard Business School Press, Boston, Massachusetts, 1992

Blank page.

CHAPTER 10
Addendum A

COTS Integration and Support Model

©Copyright Loral Federal Systems Manassas December 1994 *[reprinted with permission]*

Carolyn K. Waund
Loral Federal Systems-Manassas

ABSTRACT

Programs requiring high use of commercial-off-the-shelf (COTS) hardware and software are becoming more prevalent in the federal marketplace. Much of the emphasis on COTS solutions is due to increasing focus on information systems technology as we seek to re-engineer the government. Information systems technology is rich in COTS products and highly competitive, thus making powerful solutions feasible.

Loral Federal Systems (LFS) has in recent years moved to address this important systems integration market. Individual programs have achieved varying degrees of success in adapting traditional system development processes and management practices in the high-COTS environment. This program experience is a key resource. This paper highlights the results of an LFS initiative to use lessons-learned on recent COTS-based programs and defines a COTS integration and support model for guiding future programs. The initial model will be refined and matured by the LFS organization in the future. This paper provides: (1) observations about the current state of COTS integration, (2) a description of a model for a COTS integration and support process, and (3) a discussion of COTS program lessons-learned and their incorporation into the COTS process.

OBSERVATIONS

Our observations about the current state of large-scale COTS integration and support are illustrated in the following graph. Note that the graph compares a *"traditional"* development program using few COTS products with three variations of high COTS content programs. We observe an important reduction in effort for high COTS, yet believe that more can be achieved. The *"dream"* program requires little effort to integrate and support a system. This may be possible today for a small, single computer system, but will not likely be achieved for the large distributed systems which serve an enterprise.

LFS has identified a number of COTS product characteristics which must be dealt with effectively to drive program effort to the *"achievable"* COTS level. The COTS product *"facts of life"* are that:

- They are not interoperable with other COTS,
- Their literature overstates their capability,
- They never exactly match users needs,
- Unique versions are costly,
- Upgrades are frequent and asynchronous,
- There is limited support for previous versions, and
- They are not Ada friendly.

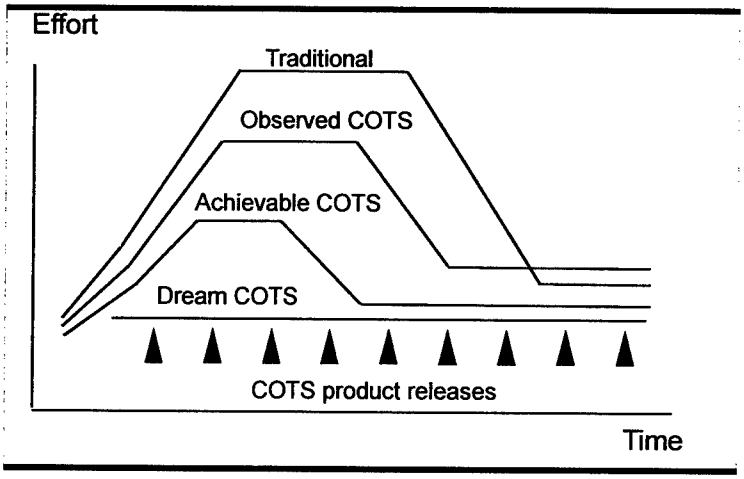


Figure 10-10 COTS Integration and Support

The COTS product releases shown under the graph represent a primary characteristic of COTS products: they change frequently in response to the demands of the commercial marketplace. These changes begin to effect a COTS-based program when the products are first selected for system inclusion, and the effects continue throughout the system lifetime. The uniqueness of COTS-based programs is the inability to control the evolution of the commercial products which make up the system. Hardware product technology becomes outdated and the old products and parts reach end of life and are no longer available. Equivalent replacement products and parts may also not be available. Software products are regularly enhanced, correcting problems, and adding and repackaging functions. Support for back level versions is often not available; COTS customers are encouraged to incorporate the upgrades. This dynamic environment indicates the need for a continuing engineering analysis to refresh the COTS-based system.

PROGRAM MODEL

The LFS approach has adapted a traditional, proven program model to one that supports a high content of COTS hardware and software. The model (below) includes both technical processes and management practices. Actual program adaptations of the traditional model were analyzed. In some cases, the adaptations were successful and are retained in the current LFS model. In other cases the adaptations failed, necessitating a return to traditional wisdom.

The model for COTS integration programs is illustrated in the following figure. It includes seven traditional technical processes performed in each iteration of activity. A concurrent engineering team, addressing all disciplines, is active throughout, coordinating all technical activities. Seven traditional management processes provide program direction, evaluation, and control. Iterations are grouped into program acquisition phases, beginning with pre-proposal and ending with operations. Contract start is at the beginning of the development phase. In each iteration, the emphasis of technical activities varies with the phase and objectives. A knowledge base captures information from all LFS division programs for use across the organization.

Each iteration begins by establishing objectives, based on the results and evaluations of prior iterations. Technical activities are scheduled with sub-objectives which, when completed, achieve the iteration

objective. Each iteration ends with an evaluation of progress against iteration objectives and program goals. The evaluation includes actual and estimated costs and a revised risk assessment, along with recommended actions and objectives for subsequent iterations. These direction and evaluation activities are the key program control mechanisms. This program model is suitable for COTS-based system integration and support because it:

- Takes advantage of COTS product availability,
- Includes prototyping iterations to reduce risk,
- Recognizes that much engineering work is performed prior to contract award, and
- Accommodates COTS product upgrades asynchronous to the program schedule.

The table indicates characteristics of each phase and the integration objectives of iterations within each phase. The following paragraphs provide additional description.

For COTS programs, the pre-proposal iteration(s) determines the feasibility of satisfying program objectives and high level system and support requirements with products which can be commercially available in the needed time frame. This iteration emphasizes requirements analysis and system architecture tasks. Customers, systems integrators, and COTS vendors interact informally or via Requests for Comments (RFCs) or Requests for Information (RFIs) during this time frame. The hardware and software product selection tasks are characterized by identifying that suitable options exist. Initial product selection is accomplished. Product experience or hands-on product evaluations are essential. Relying on product documentation or demonstration is an invitation to failure. The pre-proposal iteration produces the system integrator's initial system design. Results can be shared with the customer, for potential inclusion in the Request for Proposal (RFP).

The proposal phase iteration(s) adjusts the system design to RFP requirements and completes the hardware and software product selection. This iteration continues hands-on product evaluation and, if time permits, begins to integrate a prototype of the system. It is important to select the most challenging aspects of the implementation for prototyping priority. Early integration must focus on areas where the return in terms of risk reduction is the greatest. If integration problems arise, it is easier to accomplish a product change-out or architecture change in this time frame than in a later one.

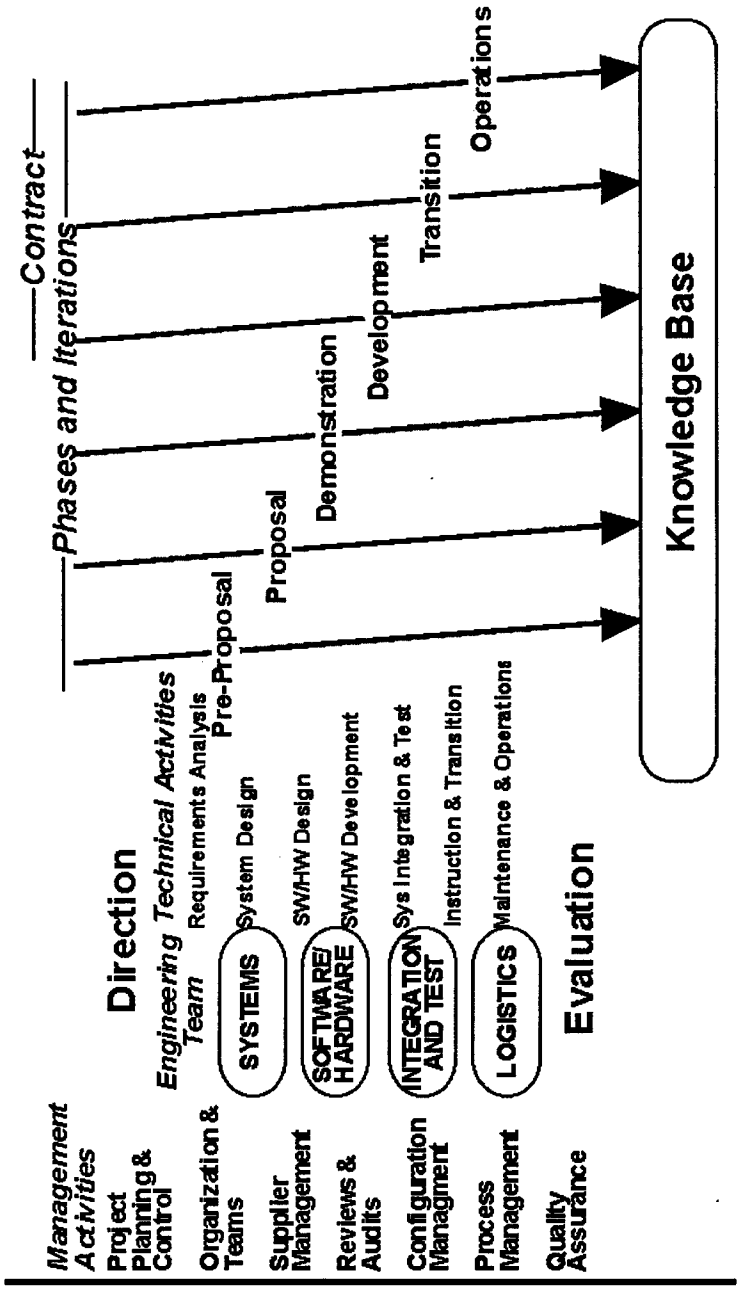


Figure 10-11 Project Model

Program Model Phases and Characteristics

The demonstration phase can significantly reduce the risk in the requirements baseline. This iteration(s) firms up the hardware and software product selection. In demonstrations, customers can evaluate the human/computer interface, explore the process and data model, and assess consistency with the operations process. If time permits, prototyping can integrate the selected products in vertical and horizontal dimensions, creating a fairly complete system implementation.

Contract work begins in the development phase. Early formal reviews serve to baseline the system requirements and design, including the COTS product selection. This is an opportunity for the systems integrator to suggest requirements changes which will increase COTS content and reduce development content on the program, thereby reducing cost and risk. Prior to this review, the pre-contract customer evaluation prototype will be reviewed for requirements and/or design change. In this way, the results of prototyping activities are fed back into the requirements, which should remain flexible until the design validation is complete. System capability and functionality are developed during the remaining development phase iterations, which may, if necessary, incorporate COTS product updates. Formal testing culminates each iteration. On the final iteration, the system is ready for customer acceptance.

In the transition phase, the initial iteration produces a fully integrated and tested system configuration installed at a customer test or evaluation site. After the evaluation period, additional iterations produce system configurations at operational customer sites or platforms.

In the operations phase, iterations are driven by requirements changes, problem fixes, technology insertion, and COTS product upgrades. Depending on the significance of the changes, an iteration may be scheduled for prototyping prior to full implementation of the change. Note that COTS-product-induced perturbations are not exclusive to operations phase. They may occur in any of the preceding phases, and need to be handled when they occur.

The COTS model features early iterations through requirements, design, and prototyping tasks, encouraging requirements modification to achieve program goals and minimize risk. This iterative process assures that the final requirements are consistent with COTS content goals. If, for example, the goal is to build the system with currently

available COTS products, it is important that the requirements reflect existing product capabilities. Appropriate adjustments to requirements can be made after COTS products have been identified, evaluated, integrated, and used in a prototype application.

PROJECT PHASE	CHARACTERISTICS	ITERATION OBJECTIVES
Pre-Proposal	• Customer/industry interaction • Request for Comments (RFC)/ Request for Information • Draft Request for Proposal (RFP) • Initial system design	• COTS feasibility • Hands-on product evaluation • Initial product selection • First integration • Initial operational analysis • Architecture definition
Proposal	• Formalized customer/industry interaction • Request for Proposal (RFP) • System design adjustment • Program planning	• Hands-on product evaluation • Integration prototype • Big product finalization • Cost model developed
Demonstration	• Formalized customer/industry interaction • Live Test Demonstration (LTD) • Design validation • Best and Final Offer (BAFO)	• Architecture validation • Integration prototype • Capability demonstration • Final bid product selection • Cost model refined
Development	• Contract starts • Extensive customer interaction • Complete implementation • Formal test	• Extensive operational analysis • Business process reengineering • User interfaced prototype • Full integration • Phasing of capability • Acceptance
Transition	• Test and Evaluation • Installation • Replication	• Evaluation site integration • Test site integration • Operational sites integration • End user training
Operation	• Engineering Change Proposal (ECP) • COTS end-of-life and update (may occur in earlier phases)	• Prototype of changes • Integration of changes

Table 10-13 Program Model Phases and Characteristics

LESSONS-LEARNED

To formulate the COTS program model, the following key lessons-learned on previous programs were addressed. For each lesson, we indicate how it is incorporated in the COTS program model and approach. In conclusion, we note barriers or challenges to incorporating these lessons, acknowledging that managing COTS programs is an exercise in tradeoffs; there is not always a single *"best"* answer!

Lesson 1: *Most shortcuts through the traditional systems development process have proven faulty, indicating a need to return to a disciplined, but tailored process.*

The COTS program model is adapted from traditional, proven LFS technical processes and management practices. Activities for COTS have been adjusted, scaled down or up, but not eliminated. For example, the traditional software development activity was redefined as COTS software product installation and customization. This requires significantly less effort than developing the product functionality from scratch, so it is a *"scaled down"* activity. On the other hand, our approach features extensive integration prototyping, so integration is a *"scaled up"* activity.

The COTS program model features close coordination of technical activity among a multi-disciplined engineering team. Key team skills include systems engineering, software engineering, integration and test, and careful consideration of logistics concerns and the views of the end user, acquisition officials and vendors. This team effort facilitates communications and permits less formality in documentation and reviews. Team agreement, based on informal preliminary documentation of requirements and design, is generally sufficient to establish technical baselines. With rapid turnaround in system development (relative to traditional non-COTS development programs), rapid decision-making is critical. This multi-disciplined team is essential.

Lesson 2: *Some requirements should remain negotiable until COTS system design is validated via prototyping.*

To reduce program cost and risk, requirements which drive unique non-COTS development and are not essential for the system's success should be candidates for change. The COTS program model uses

iterations to formalize the feedback from system design, COTS product selection, and prototype integration activities to aid requirements analysis. This feedback can specifically identify the requirements which are not capable of being satisfied using currently available COTS products. These are the requirements which drive program-unique development of product enhancements, including *"glue"* code to fix interoperability problems or selection of a high risk product. Some of the COTS product non-compliance issues and interoperability problems are discernible without prototyping. However, some issues are uncovered only during hands-on product evaluation and integration prototyping.

The COTS program model features a number of iterations prior to the baselining of requirements in the development phase. In the pre-proposal phase, the integrator should provide feedback to the government customer, indicating those requirements which drive non-COTS content. In the proposal and demonstration phases, the ability to communicate with the customer is limited. During these phases, the integrator builds up a set of recommended requirements changes which would increase COTS content and reduce program cost and risk. These are shared with the customer at contract award, for consideration prior to formal requirements baselining.

Lesson 3: *COTS product selection and system design validation should include, or carefully waive:*

- Hands-on evaluation of each product,
- Testing of each product-to-product interface,
- Prototyping of developed application-to-product interfaces,
- Testing COTS product portability to new platforms, and
- Vendor and product considerations beyond functionality.

The COTS program model features many early iterations, giving opportunity for hands-on evaluation, product interface testing, and integration prototyping. The emphasis of these activities is directed where the most benefit in terms of risk reduction can be gained. Bypassing these activities is acceptable for well-known products or previously-demonstrated interfaces, where the risk is assumed to be low. However, there can be a tendency to assume compatibility and underestimate the integration effort required.

Experience has shown that it is difficult to understand a product by talking to vendor marketing personnel or reading literature. Hands-on evaluation by the integrator permits a vendor-independent assessment of how the product meets the program requirements, and helps to avoid conflicting interpretations of requirements and what it takes to satisfy them.

Although some products are designed to interoperate, there is no guarantee that product-to-product interfaces will operate properly for the types of data to be supported or the environment of the program. Testing can determine if interface problems exist. Resolution may involve changes by one or both vendors or integrating mechanisms created by the integrator. Early problem detection can influence design and product selection and result in lower program effort and cost.

The boundary between developed applications and COTS products in the operational system can be a source of integration problems. This has been the case on a number of LFS programs using Ada. Ada interfaces or bindings to products are not as prevalent, highly-functional, or mature as those for C. If the degree of compatibility with application development tools and COTS products is overestimated by the integrator, the development effort can be larger than expected. Prototyping is indicated for unproven interfaces.

Sometimes the best COTS software product for the functional requirements has never operated on the hardware and operating system platform of choice for the program. In these cases, the integrator and the vendor agree that a product port is the best option. There is always some risk in porting a software package, especially if the product has never been ported before.

Considering product characteristics other than functionality can also be important in the decision-making process. Such aspects include current quantities in use, reliability, product support, and past performance of the vendor.

Lesson 4: *A well-defined architecture can lessen the impact of COTS product upgrades. Integration facilitators can reduce risk, but also have disadvantages.*

In the past, the tendency was to integrate COTS applications in an ad-hoc fashion, creating *glue* code as needed to permit applications to interface with each other and external interfaces. Each glob of glue is unique to the set of interfaces between a pair of applications, and may require frequent modification as business needs and technologies change. Because this type of system is expensive to build and support, other mechanisms are preferred.

The architecture for high COTS content systems needs to consider a number of approaches which can facilitate the integration of both the COTS products and developed components that comprise the system. These approaches include architecture definition, standards, and frameworks, which are discussed in the following paragraphs.

The major step in creating a top level architecture determines the degree of integration needed among system components and establishes an integration strategy, which may feature using a number of integration mechanisms.

When integrating COTS and developed system components, there are five dimensions of integration to consider:

- **Data Integration**. Data created by one component are transformed and transported to another component for its use, in the format and context it requires.
- **Control Integration.** An activity or product of one component will cause the activation of one or more other components.
- **Process Integration**. The enterprise goals are translated into processing and storage decisions for all the components that participate in that goal. These decisions can influence control integration and data handling. *"Workflow"* is an implementation of process integration.
- **Presentation Integration**. The user interfaces have the same look and feel.
- **Platform Integration.** The COTS products and developed software components are independent of the platforms and operating systems inherent to those platforms. A heterogeneous system can result.

The required degree of integration in each of these dimensions is determined to influence the evaluation criteria for selecting the integration mechanisms, COTS products, and defining an architecture to isolate and minimize change as COTS products change.

Standards-based COTS product interfaces facilitate integration, but do not guarantee compatibility. Mature standards are unambiguous but tend to be complex, while immature standards still allow implementation flexibility. This permits different interpretations by vendors which can cause interface problems.

Integration framework products, such as those being used in software engineering environments, provide tool integration services which show promise of facilitating COTS application product integration. Usage of these framework products in this domain is not yet proven, but the integration framework concept and services constructs can be effective tools in system design.

Selecting a product suite from a single vendor, vendor partnership, or coalition is a low-risk integration approach which minimizes problems within the suite. Unfortunately, there may be no flexibility to add a best-of-breed product from outside the suite due to the closed nature of the design.

Product characteristics such as published Application Programming Interfaces (APIs), which reveal most product functions, many documented user exits, and source code availability, can facilitate product adaptation by the integrator to the system environment. The use of these characteristics, however, implies software development.

In the COTS program model, iterations through system design, product selection, and prototyping activities permit the evaluation of a variety of strategies for COTS-based system architecture and design. Various integration facilitators and mechanisms can be explored.

Lesson 5: *Significant advantage can be gained by reusing previously integrated solutions.*

There is value in reusing previously integrated COTS solutions to reduce program cost and risk. As a group, the LFS divisions have in place a number of experience-sharing mechanisms which facilitate this type of reuse. Although LFS programs address a variety of information systems application domains, most share a need for

integrated platform services (including networking, distributed computing support, and other middleware functions). Standards-based integrated solutions in this area are particularly good candidates for reuse.

In some cases, an integrator can string together contracts with similar needs and reuse integrated solutions or solution parts. To extend the set of ready-to-use solutions, the integrator can invest in COTS product integration initiatives.

The COTS program model includes a knowledge base of product and integration experience that is contributed to by each contract or investment program and is available for use by all programs.

Lesson 6: *Vendor contracts are unique, complex business/technical/legal agreements which must clarify all requirements and expectations.*

LFS has found that COTS vendor contracts must go beyond standard commercial license and services agreements, and become similar to major development subcontractor relationships. This applies when standard COTS offerings do not meet program requirements and must be enhanced. As a result, COTS vendor relationships become more complex than that of simple commercial product or service offering procurements.

These unique relationships are also more difficult to negotiate and manage because COTS vendors do not typically deal in this manner. Their business and technical processes are tuned to produce and support a standard offering, not to provide specialized solutions for a particular customer's system. Another contributing factor to negotiation complexity is the fact that many commercial vendors are not experienced in dealing with the Government. Unfamiliar military contracting concepts and terminology, in particular, can contribute to major vendor misunderstandings.

For each COTS program, LFS uses an experienced supplier manager who reports to the program manager. The supplier manager leads a multi-disciplined team which defines and negotiates vendor contracts and manages vendor contract performance.

Lesson 7: *Plan and budget for frequent and uncontrollable COTS product end-of-life and update events during all phases of the program.*

The uncontrolled nature of COTS products is often not recognized and planned for. When not properly planned, significant cost impacts can occur. It is likely that after system delivery, COTS software product upgrades will be released by vendors and hardware products and parts will become unavailable. We recommend planning for the engineering effort required to ensure system integrity.

The COTS program model features continuing iterations in which COTS product updates can be scheduled for integration. The model handles each COTS product end-of-life or upgrade event as an engineering change proposal (ECP). Notification of a product end-of-life event by the vendor is followed by impact assessment, selection of a replacement product, and planning for its integration into the system. Notification of product upgrade initiates impact assessment and integration planning.

Lesson 8: *Cost estimation methods for COTS integration programs must be improved.*

Industry-proven cost estimation models exist for traditional development programs with a high content of software development. These models have been calibrated with the results of many programs throughout the years. This is not the case for programs with high COTS content, where estimation models do not exist.

To increase the predictability of COTS programs, LFS has standardized the process on COTS programs and developed a program cost estimation model tailored for COTS integration. LFS will use this cost model to estimate new programs. Metrics collected from previous programs are being used to calibrate the model.

CHALLENGES

Many of the actions which must be taken to remedy the problems experienced on COTS programs involve a shift of effort into the pre-contract timeframe. This is challenging to systems integrators because it stresses the Bid and Proposal (B&P) budget in a constrained timeframe when customer communications are limited.

For example, prior to submitting the offer to the customer, it is often necessary to validate a significant part of the COTS system design via prototyping to reduce risk. This effort is costly, yet without this validation, the schedule and cost risk of the program under contract is increased, and may be intolerable.

It is highly desirable to adjust requirements based on discoveries during system design, product selection, and prototyping, yet this is not accommodated by the procurement process. RFP requirements are mandatory and inflexible during the proposal and demonstration phases when system design, product selection, and prototyping are accomplished. Requirements changes which could increase COTS content and simplify integration are not allowable in the pre-contract timeframe, and may be difficult to incorporate after award on a fixed-price contract. This may add unnecessary cost and risk to the program.

A large part of the challenge on COTS programs comes from a lack of widespread understanding and experience with the unique aspects of COTS programs in government and industry. General agreement has not yet been achieved on the complexity of designing and supporting a system made up of uncontrollable, commercially-driven elements.

CONCLUSION

Improved COTS program predictability will benefit both government and industry. This COTS program model is representative of many challenging COTS programs and establishes a framework for assembling processes tailored for COTS. LFS is seeking process maturity by continuing to refine the model and standardize effective processes for COTS integration and support.

About the Author

Carolyn K. Waund is a Senior Programmer in the Federal Systems Integration Laboratory at Loral Federal Systems in Manassas, Virginia. She is currently responsible for defining methods and architectures which facilitate COTS product integration and software engineering. Prior to its acquisition by Loral, she worked for IBM Federal Systems Company in Houston, Texas for 29 years. She has been involved in a number of systems integration programs, both for federal and commercial customers. Most of her career has been focused on *real-time* support of manned spaceflight, performing both systems and software engineering tasks for NASA Johnson Space Center. She received her B.A. in Mathematics from the University of Texas at Austin.

CHAPTER 10
Addendum B

Rate Monotonic Analysis: Did You Fake It?

NOTE: This article is found in Volume 2, Appendix O, *Additional Volume 1 Addenda.*

Blank page.

CHAPTER 11

Software Support

CHAPTER OVERVIEW

*In **Operation Desert Storm** the intensity of battle coupled with large forces using Information Age weaponry and communications created the most intense electronic battle field ever witnessed. The **E-3 AWACS** was an integral part of the battle serving as the "eye" that tracked all battle space aircraft and directed interceptions while safeguarding our forces from surprise enemy aerial attack. The overwhelming density of diverse electronic signals transmitted and received created such a congested environment that the E-3s' full mission capability was greatly hindered. This E-3 problem had to be quickly corrected and a dedicated software support team sprung into immediate action. The E-3 radar software was rapidly revised, flight tested, and on its way to deployed aircraft within 96 hours. This quick reaction, modification, and change-out during the heat of battle emphasizes the operational necessity for easily supportable software..*

*The ability to continuously support our major software-intensive systems is a paramount mission requirement. **Supportability** is critical because there is always a inevitable need to correct latent defects, modify the system to incorporate new requirements, enhance the existing system to add capability, and alter it to increase performance. The ability to accommodate change is an integral part of major software-intensive systems requirements.*

Unfortunately, when we have fielded insupportable systems, we have often had to expend the considerable time and funds necessary to provide the required support or we have had to abandon them altogether. We learned that it is far more cost-effective to address supportability as we define requirements, design the system, and plan for its operational life. In this chapter you will learn how to reduce the risk of acquiring, managing, and maintaining software-intensive systems by ensuring that they are modifiable, expandable, flexible, interoperable, and portable — i.e., supportable.

***Software support**, often called re-development, addresses the maintenance life cycle phase where major software costs occur. Support planning addresses the development acquisition and entails RFP development that provides for delivery of full documentation, data rights, and delivery of the software engineering environment (SEE) used by the developer.*

CHAPTER 11 Software Support

When tasked with maintenance responsibility of legacy software which has become technologically obsolete, has deteriorated through years of changes, or must be changed anyway to work with new hardware or other software, it may be cost effective to ***re-engineer*** *it. This involves systematic evaluation and alteration of an existing system to reconstitute it (or its components) into a new form or converting it to Ada to perform within a new operational environment, to improve its performance, or to reduce maintenance costs. This process can combine several subprocesses, such as reverse engineering, restructuring, re-documentation, forward engineering, or retargeting. [NOTE: For additional information on F-22 PDSS efforts, see Volume 2, Appendix K. Software Support.]*

CHAPTER 11
Software Support

A TOTAL LIFE CYCLE APPROACH

With the exception of the **B-2 bomber**, DoD will not be purchasing any additional bomber aircraft and procurements of new, advanced fighter aircraft [i.e., the **F-22** or **Joint Advanced Strike Technology (JAST)**] will not occur until after the turn of the century. Thus, we have to rely on existing aircraft platforms for several years. The current modification to the **B-1B Lancer** is a prime example of this. As mentioned in Chapter 1, *Software Acquisition Overview*, the B-1B is being upgraded to a conventional munitions capability. The bulk of the effort focuses on the enhancement and modification of the B-1B's offensive avionics software component. ***These trends indicate that the future capability of our major software-intensive systems is inexorably dependent on our ability to cost-effectively maintain them.***

Software support (or maintenance) is really a poor name for the post-deployment software support (PDSS) activity. In other engineering contexts, maintenance implies repairing broken or worn-out parts. But software does not break — nor does it wear out. It is for this reason that PDSS is often called the ***re-development*** phase. As defined by the IEEE, **software maintenance** is the

> *Modification of a software product after delivery to correct faults, to improve performance, or other attributes, or to adapt the product to a changed environment.* [ANSI/IEEE83]

Software is alive! Whether it is in production or not, it is always in the process of becoming, evolving, changing. Research on software maintenance shows that user requirements impacting software account for **41%** of post-deployment support costs, while hardware changes

account for only **10%**. [BASSETT95] That is to say, over half of all software support is driven by changes in the system's external environment. Because software must evolve in response to its external environment, it is more like a living thing than an inanimate object that only needs to be designed once, and thereafter, infrequently repaired or maintained. With software, development (and re-development) is the norm in response to external changes. Therefore, designing for maintenance must be incorporated and unified with development.

Software support is both different from and the same as development. It is different because the developer has no existing system from which to work; whereas, the maintainer must be able to read and understand already existing code and solve problems within an existing framework which constrains the solution set. The developer has no *product knowledge* because the product does not yet exist. The maintainer must have complete product knowledge to do his job well. Support is the same as development because the maintainer must perform the same tasks as the developer, such as define and analyze user requirements, design a solution (within the constraints of the existing solution), convert that design into code, test the revised solution, and update documentation to reflect changes. Figure 11-1 illustrates how support tasks correspond to and mirror the development process. [GLASS92]

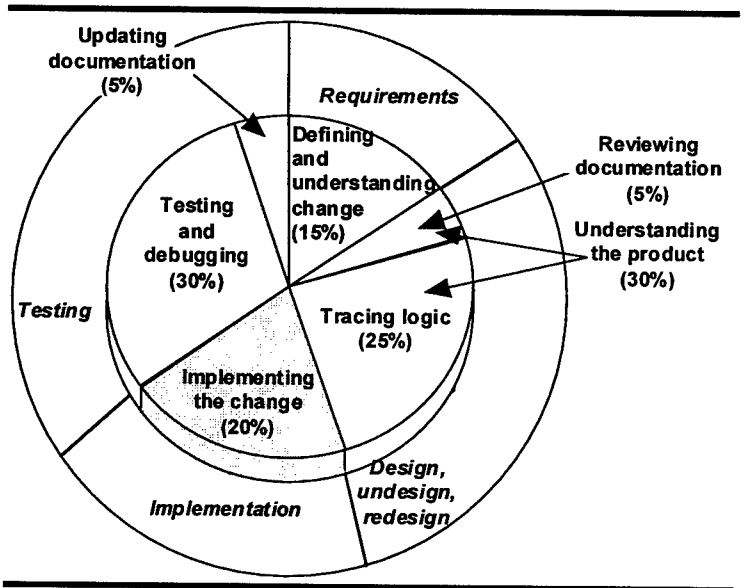


Figure 11-1 Support Tasks Superimposed on the Software Development Phase

Software Support Cost Drivers

Nothing will throw an infantry attack off stride as quickly as to promise it support which is not precisely delivered both in time and volume.
— Brigadier General S.L.A. Marshall [MARSHALL47]

The demand for precisely delivering software support in time and of high quality has never been greater. However, software support is by far the biggest life cycle cost driver and the most significant source of system risk for all major DoD software-intensive acquisitions. Although software support actually occurs during the post-deployment phase, it must be planned for upfront during requirements definition and design. It must also be budgeted for and continuously addressed throughout the system's life. ***Developing supportable software is one of the most important criteria for software success.*** All the causes of cost and schedule overruns, performance shortfalls, and for programs being thrown off stride, as discussed in Chapter 1, *Software Acquisition Overview*, are amplified once the system is in the hands of the maintainer. Therefore, the ***Software Crisis*** has really been the ***Maintenance Crisis***. According to numerous DoD and industry studies, the typical cost to maintain a software product is from ***60% to 80% of total life cycle costs***. Your challenge is to minimize the cost of software maintenance, and to avoid being at the heart of the *Crisis*. These costs are depicted on Figure 11-2.

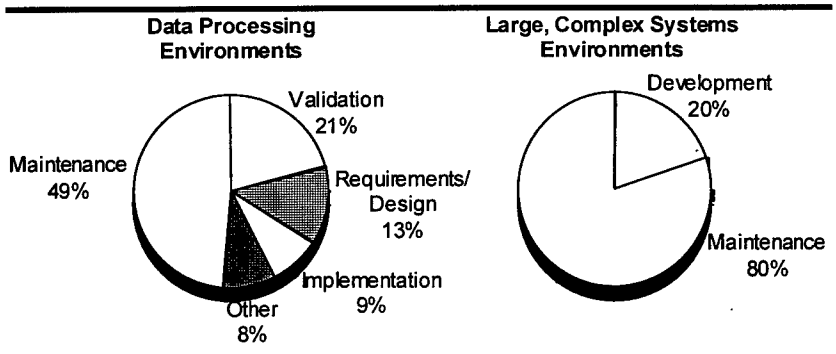


Figure 11-2 Life Cycle Support Costs

These cost increases during the software maintenance phase have historically been caused by dramatic ***decreases in productivity*** (measured in LOC/manmonths or function (feature) points/ manmonths.) Productivity drops of **40:1** have been reported during

software support. [BOEHM81] For example, what cost $150/LOC to develop might cost $1,000/LOC to maintain. This significant increase in system cost demands that basic decisions about how the software will be maintained are made during the concept and design phases. Easy access to the software and an inexpensive medium for distributing enhancements can have significant effects on life cycle costs. A well thought out concept of operations includes hardware provisions for spare connectors, card slots, and memory capacity to facilitate interoperability to new software systems as they are fielded and integrated into the defense inventory. [PIERSALL94] A flexible, modular **architecture** is also essential for ensuring ***understandability***, ***modifiability***, ***interoperability***, ***reusability***, ***expandability***, and ***portability*** — all prerequisites for supportable software.

Software Support Activities

Based on a study of 487 commercial software development organizations, Figure 11-3 illustrates how software support changes are distributed among support tasks. Most software support dollars are spent on defining, designing, and testing changes. After these activities are performed (whether there is one unit or hundreds of units in the field), subsequent increases in cost are marginal. Support activities include:

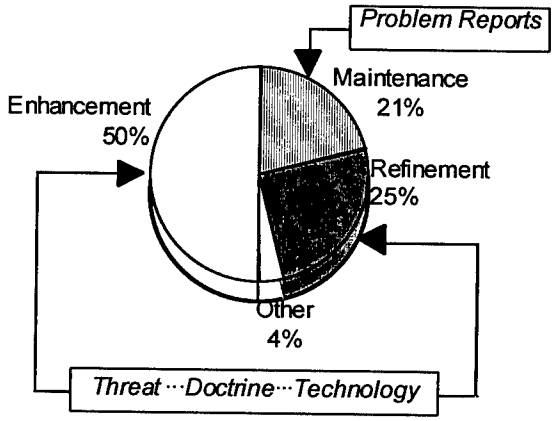


Figure 11-3 Causes of Software Changes [PIERSALL94]

- Interacting with users to determine what changes or corrections are needed,
- Reading existing code to understand how it works,

- Changing existing code to make it perform differently,
- Testing the code to make sure it performs both old and new functions correctly, and
- Delivering the new version with sufficiently revised documentation to support the user and the product.

During operational testing, supportability evaluations concentrate on software code, supporting documentation and implementation, computer support resources, and life cycle process planning. Due to its impact on software support, spare computing capacity is also examined. The four areas the **Air Force Operational Test and Evaluation Center (AFOTEC)** evaluates for supportability are illustrated in Figure 11-4. For example, maintainability evaluations consist of questionnaires that concentrate on the specific characteristics of a maintainable system, such as consistency, modularity, and traceability. Software supportability is evaluated by the developer when the documentation and source code are initially baselined (usually during initial integration test and evaluation) and then periodically until the completion of software development. The information gained during integration testing helps the developer build more maintainable software.

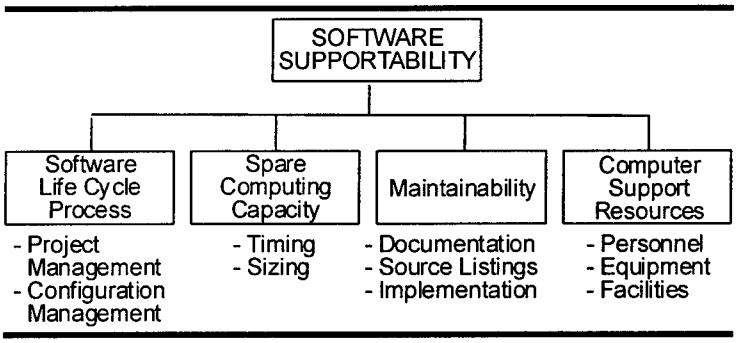


Figure 11-4 AFOTEC Software Supportability Evaluation Areas

Software Support Issues

In theory, ***software never wears out!*** It has none of the physical properties found in hardware upon which the forces of Nature and the operational environment can play that cause physical systems to decline in performance. When hardware begins its life span, it often has a high failure rate (defects per unit time) until manufacturing defects are ironed out. The failure rate then drops to an acceptable

low-level where it remains (often for many years) until components begin to wear out. At this point, the failure rate begins to climb again. This trend, called the *"bathtub curve"* by hardware engineers, is true for all hardware systems — whether an automobile, a radio, or a computer.

While software does not wear out in the physical sense, ***it does deteriorate!*** There is an astounding difference when the software failure rate is superimposed on the bathtub curve. Like hardware, new software usually has a fairly high failure rate until the bugs are worked out. At which point failures drop to a very low level. *[A significant exception is software developed using Cleanroom techniques (discussed in Chapter 15, Managing Process Improvement) where initial failures are also low.]* Theoretically, software should stay at that low level indefinitely because it has no tangible components upon which the forces of the physical environment can play. However, after software enters its operational life (during **PDSS**), it undergoes changes to correct latent defects, to adapt to changing user requirements, or to improve performance. These changes make the software failure rate curve steadily begin an upward journey. Hardware deteriorates for lack of maintenance, whereas ***software deteriorates because of maintenance!*** [GLASS92] By making changes, software maintainers often inadvertently introduce ***"side-effects"*** causing the defect rate to rise, as illustrated in Figure 11-5.

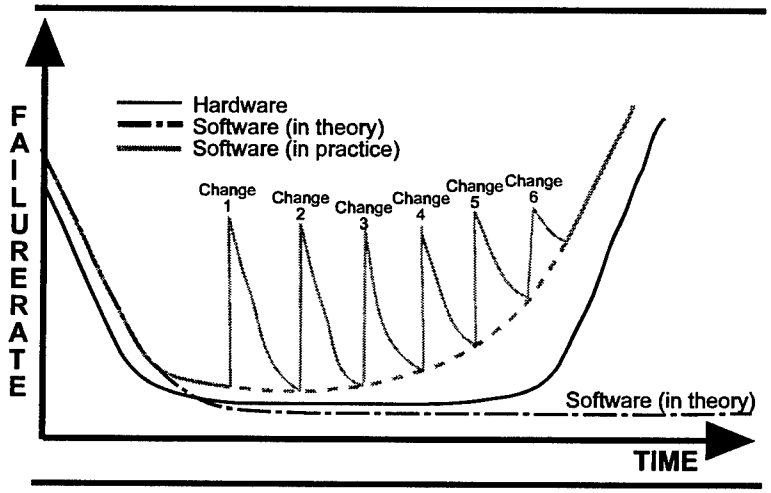


Figure 11-5 Bathtub Curves for Hardware and Software

Although side-effects can be quite complex, most are caused by one thing—***there are no spare parts for software!*** When software fails the part causing the failure cannot simply be replaced with a spare. When software fails, from defects inserted during maintenance, often the only way to correct for the cause of failure is through design modification. Every time the design is modified it weakens the original structure (or how the modules work internally and with each other) and eventually the software begins to fall apart. Undisciplined maintenance (or that performed in the field under stressed conditions) frequently compounds the problem. Maintainers, struggling against time to make corrections, modifications, or adaptations to new requirements, often compound the defects created by the last generation of maintainers. In the rush to get the product to impatient users, they take short cuts—exacerbating the software's deterioration. Problems also arise when there is a failure to modify the design when patches are made (causing the design and code to be out of synch), a failure to update documentation, or a failure to use modern concepts of design and programming in initial development.

Most of the problems associated with software support can be directly traced back to deficiencies in the way the original software product was planned, managed, and designed. ***Lack of sound software engineering discipline, control, and attention to the design of modular software architectures during development translates into software support problems resulting in excessive maintenance costs.*** Some classic software support issues include:

- Lack of requirements traceability;
- The evolution of software versions or releases that are difficult or impossible to trace *[the evolution of changes that are not documented];*
- A difficult or impossible to define software development process;
- Impossible to understand code *[software understandability increases as the number of software configuration items increases];*
- Documentation that is nonexistent or of such poor quality that it is useless *[documentation must be understandable and consistent with the source code to be of value];* and
- Inflexible software not designed to accommodate change *[unless the architecture allows for change, modifications to the software are difficult and defect-prone].* [PRESSMAN92]

This last point is, perhaps, the most critical deficiency. The software architecture should carefully address abstraction, encapsulation, and information hiding *[see Chapter 5, Ada: The Enabling Technology,*

"Ada Program Unit," and "Ada's Packaging Feature"] to minimize dependencies. By separating computational and operational details from interface calls, and by maximizing use of object-oriented design, the software can be easily modified. Modifications can occur during development and during post-deployment operation with less risk of introducing unwanted side effects.

Many factors play in the supportability equation. An undisciplined, poorly managed development process where design, coding, and testing were conducted with inadvertent carelessness negatively impact the support task. **Design characteristics** that affect software supportability include: design complexity (including related attributes of software size, structure, and interrelationships); stability and flexibility of the design itself; adequacy of documentation to support PDSS; completeness of the software development effort; and the extent and implementation of configuration management practices for both operational and support software. [SHUMSKAS92] Other factors within the development environment that impact software supportability include:

- Availability of qualified software personnel,
- System structure understandability,
- Ease of system handling,
- Use of standardized programming languages,
- Documentation structure standardization,
- Test case availability,
- Built-in debugging mechanisms,
- Delivery of the original development SEE to the maintenance organization, and
- Availability of appropriate computer hardware to conduct maintenance activities. [PRESSMAN92]

COTS Software Support Issues

Software support includes support of government-developed software, contractor-developed software, and **commercial-off-the-shelf (COTS) software** *[discussed in detail in Chapter 13, Contracting for Success].* Issues to consider when supporting COTS software include:

- The acquisition agent must acquire appropriate documentation and data rights, licensing, and subscription services (such as options to purchase or escrow proprietary information) which allows the

Government to support the software if contracted support becomes unfeasible.

- The software support activity (SAA) must maintain appropriate licensing and subscription services (vendor field change orders and software releases) throughout the life of the system.
- ***COTS resources must not be altered so as to preclude contractor logistics support or void licensing or subscription services.***
- The supporting command must provide logistics support and contract for subscription services required to update and maintain COTS assets. It must also evaluate operational and logistic impacts of change due to subscription-related hardware and software upgrades.
- The operating command must provide a technical review of proposed changes during upgrades and changes to COTS assets. It is responsible for evaluating effectiveness and mission impact of changes due to subscription-related software upgrades.

PLANNING FOR SUPPORT SUCCESS

In recent years, early planning for software support has become a main DoD acquisition priority. Learning from costly past mistakes, the early **F-22 Advanced Tactical Fighter (ATF)** *[now the F-22]* planners wanted to make their weapon system a "*maintenance man's dream*," according to **Colonel John Borky**, former director of ATF Avionics. [BORKY90] **Colonel Ron Bischoff**, Air Logistics Center (ALC) system program manager for the F-22, remarked, "*We are practicing* [with F-22 support and maintenance design] *what we always said we were going to do, but never did...*[Before] *it was a build it, then fix it, way of doing business.*" [BISCHOFF91] In the past, the system program manager responsible for supporting the aircraft was not assigned until late in the design process. Support problems were not addressed until after an aircraft was deployed and maintenance problems occurred. ***ATF planners specified support requirements early***, which then became part of the RFP. Colonel Bischoff explained that planning for support success was accomplished by making it ***a source selection criterion that support issues be addressed during the design stage.***

Colonel Bischoff remarked that writing and maintaining software for the F-22 will be a much larger task than for any other aircraft in history. He explained, "*The F-22 leads DoD's list of the most complex software projects, with a projected 7 million lines-of-code.*" [BISCHOFF91] ATF planners enforced consistency and

completeness by mandating the use of Ada for all ATF software systems. By using Ada, all F-22 software engineers are forced to use common terminology, from ground support systems to operational flight programs. Bischoff claims, "*That was a major step forward. Ada makes the software much more supportable because it is written in much clearer text. Lack of documentation killed us in the past.*" ATF planners also enforced the use of a **common Ada software engineering environment** that provides uniform development tools for all the software development team members.

To augment F-22 support success, Air Force and contractor personnel will work together as IPTs to maintain F-22 software. To plan for this requirement, the ALC F-22 system program office (SPO) has ALC software personnel involved *shoulder-to-shoulder* with contractors so they will understand what is being done and why. Colonel Bischoff boasted, "*We're already planning for the first update to the operational flight program within a year or two after the first F-22 rolls off the production line!*" [BISCHOFF91]

As discussed above, decreases in productivity during PDSS can be tied to ***increases in software complexity*** the longer it is in the support phase. The more modifications made to the software (especially to a poorly engineered product), the more complex it becomes with corresponding increases in the introduction of defects. These exponential increases in effort (and cost) are mainly the product of ***poorly engineered software***. [PRESSMAN92] Therefore, ***planning for supportability upfront is a major determinant of software development success***. Software, not developed with maintenance in mind, can end up so poorly designed and documented that total re-development is actually cheaper than maintaining the original code. With today's shrinking defense dollars, ***failure to make software maintenance a design priority*** would not only be poor management on your part, but could very well result in an inability to support your product. In fact, the need for good software maintenance planning is so crucial that the **Ada language** *[discussed in Chapter 5, Ada: The Enabling Technology]* was designed specifically to make software maintenance easier. Some desirable software attributes for supportability are found in Ada's **package-body separation** and **generics**, in addition to **inheritance** and **dynamic binding** found in **object-oriented design (OOD)** methods (available in Ada 95). Skill and experience in designing Ada systems is another factor to consider when addressing the supportability issue. [PDSS90]

Software Support Cost Estimation

The variety and undefined scope of future changes throughout the software life cycle make estimating support costs one of the most difficult — yet one of the most important to consider due to their impact on the DoD budget. Most software estimating models estimate software support costs; however, the types of activities, and therefore, the costs included in their estimates, vary significantly from model to model. Most parametrically-based software support estimating models provide a top-level approximation of sustaining engineering and support requirements. ***They do not produce estimates that can be reliably used alone as the basis for a software support budget or similar purpose.*** Once software has been transferred into a support environment, changes to the software (especially major changes or additions to basic software functionality) must be estimated using software models calibrated to the re-development environment. *[See Chapter 10, Software Tools, for a discussion on parametric support estimating models.]*

SOFTWARE RE-ENGINEERING

The concept of **re-engineering** is relatively new within the software development community. The motivation behind re-engineering is to get a handle on the ever-growing software maintenance burden. The rapid evolution of software and hardware technology over the past 20 years has left DoD with a legacy of millions of lines of failure-prone code, written in a conglomeration of languages, running on a hodgepodge of incompatible hardware. The re-engineering option may prove beneficial where large libraries of non-Ada code exist.

"Re-engineering" is defined as the examination and alteration of a software system to reconstitute and re-implement it in a new form. The re-engineering process involves recovering the design from an existing application and using that information to reconstitute it to improve its quality and decrease maintenance costs. While re-engineering re-implements existing system functions in a better, more efficient manner, new or improved functions are also often added. [PRESSMAN92]

Re-engineering Decision

Re-engineering of old, worn-out or obsolete code is often economically justified. The lengthy DoD acquisition process often takes a decade or more for large software-intensive systems. By industry standards, military software is often obsolete before it enters the field, at which point a 20-year operational life usually lays ahead. The cost of maintaining software over its extended life can be from two to 10 times as much as the cost to initially develop it. The decision to re-engineer software is often one based on the premise to *"pay now or pay much more later."* [PRESSMAN92] There are basically three situations when re-engineering is beneficial. These include:

- When the existing system has become technologically obsolete and must be replaced;
- When the existing system has deteriorated to the point where it has severe technical problems; and
- When it might be expedient to upgrade the existing system. [SNEED91]

You may choose to re-engineer if you reach the conclusion that it is better to *pay now*, rather than waiting to *pay-much-more-later*. *"Paying now"* is what Perry calls ***avoiding the rathole syndrome***. He defines a *rathole* as the dark place where software maintainers throw their money with no possibility of return on investment. He equates the legacy software rathole with the old car rathole. In the short-term, it is cheaper to fix your old car than it is to buy a new one. But over an extended period, the out-of-pocket expense for parts and labor to patch your old clunker will cost you more without increasing its resale value than investing in a new car. He also explains that software maintenance ratholes are like ratholes in the woods. Once you plug one up, the rat digs another. [PERRY93] Re-engineering, when cost effective, can provide you with a way to plug up all your ratholes and have a spanking new system with all the bells and whistles your user desires. It may well be the long-term, low-cost solution to your software maintenance problems. The reasons to re-engineer include:

- To reduce maintenance costs,
- To decrease defect rate,
- To convert to a better language or hardware platform,
- To lengthen the life-span, and
- To enable changes in the user's environment.

Another reason to re-engineer is often based on the logical migration of the system. Since the system has to be dramatically changed anyway, it might as well be upgraded to more current technology. Your re-engineering decision must be based on a thorough feasibility analysis of the costs, benefits, and risks involved in continued patching (if possible) versus re-development (starting from scratch) versus re-engineering. This analysis is based on a calculation of the target system's expected lifetime and the comparison of re-engineering costs with the costs of a new development. A rule of thumb is, ***re-engineering is a viable alternative when the cost to re-engineer is not more than 50% of the cost to re-develop***. It may also be determined that it is too expensive to re-engineer the entire system. [SNEED91] Studies conducted by major industry software developers indicate that ***80% of the problems are caused by 20% of the software***. [JONES91] Therefore, in some cases, only 20% of a system may need re-engineering.

Re-engineering is only one of several options you have as a maintenance manager in fulfilling your user's needs. These options must be weighed one against the other. Factors to consider, in addition to cost, include:

- The added value of re-engineering relative to the value of a new system and the value of the present system.
- The risk of re-engineering relative to the risk of a new development and the risk of doing nothing.
- The life expectancy of the existing system relative to the time required to re-engineer it and the time required to re-develop it. [SNEED91]

Re-engineering Process

Re-engineering involves a number of engineering concepts. How these engineering tasks make up the re-engineering process and relate to each other is illustrated on Figure 11-6 (below). These methods include:

- **Reverse engineering** is the process of examining an existing software system to abstract its design and fundamental requirements. It is also the end-to-end process used to understand the existing software well enough to change it. It is the opposite of *forward engineering* (the traditional way software is developed) *[discussed in Chapter 14, Managing Software Development].*

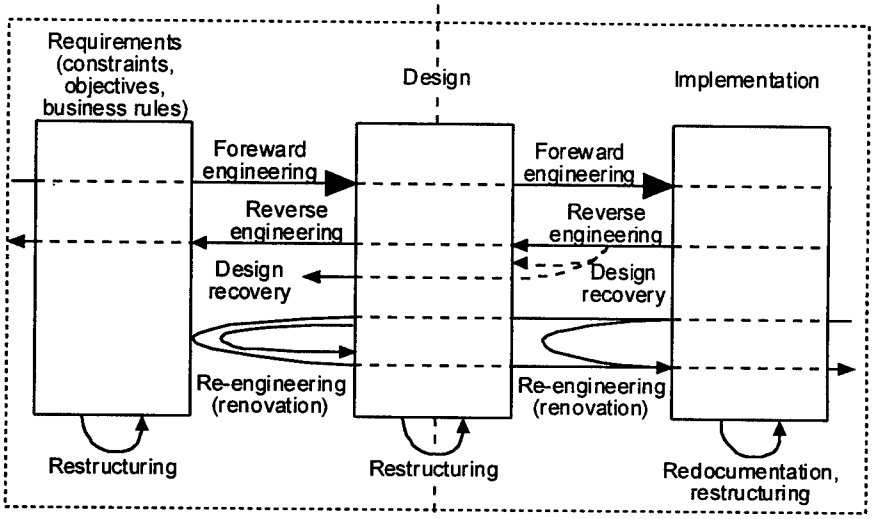


Figure 11-6 Relationship Among Support Engineering Tasks [GLASS92]

- **Forward engineering** is the set of engineering activities that use the products and artifacts derived from legacy software and new requirements to produce a new target system.
- **Restructuring** is the process of reorganizing or transforming an existing system from one representation form to another at the same relative abstraction level, while preserving the subject software's external functional behavior. Most commonly applied, restructuring involves taking (perhaps unstructured) software and adding structure.
- **Re-documentation** is the process of analyzing the software to produce support documentation in various forms, including users' manuals and reformatting the system's source code listings.

Other software support engineering concepts not illustrated on this figure include: **retargeting**, the process of transforming and hosting (or porting) existing software to a new configuration; and **source code translation**, the transformation of source code from one language to another or from one version of a language to another version (e.g., translating COBOL-74 to COBOL-85). [OLSEM93]

Your re-engineering strategy can be fruitfully integrated into your domain engineering approach. This may involve looking at re-engineering as a total migration plan which can involve a number of incremental steps — rather than as a single event at one point in time. A comprehensive model of the re-engineered system can also be

developed and maintained while the implementation of the plan is staggered as resources permit. *[For more information see Feiler's Re-engineering: An Engineering Problem, SEI Special Report, 1992.]*

Re-engineering to Ada

Ada is required for all DoD software when more than one-third of the existing code must be altered. This pertains to individual CSCIs, entire subsystems, or systems. It applies if five years have elapsed since system/subsystem acceptance or a cumulative total of one-third or more of the original source code lines have changed since acceptance. With this requirement, converting large libraries of existing code to Ada will eventually negate the need for maintenance organizations to retain expertise in languages other than Ada. Complex and costly integration and cohabitation problems that arise when hybrid systems of Ada and non-Ada code are developed, operated, and maintained will also be avoided.

If you are responsible for the long-term maintenance of any major software system not written in Ada, you should consider the significant, potential benefits of re-engineering. From a management perspective, ***automated tools are the key to re-engineering success***. Where tools are available, re-engineering can usually be performed efficiently and economically, particularly in the **MIS** world where many re-engineering products are available — with more under development.

You should be cautioned, however, that some re-engineering techniques neither produce good readable Ada code nor take advantage of Ada's good features. The goal of the re-engineering process is to retain as much of the original software function and gain as many of the benefits of Ada as economically feasible. Whether or not re-engineering is the right choice for your particular program or organization should be based on careful investigation of available tools, proven successful techniques, return on investment, cost, and risk issues. *[To assist program office and SAA personnel in determining whether to re-engineer legacy software, leave it unchanged, or re-develop it, consult the Software Re-engineering Assessment Handbook (SRAH) available from the Air Force Cost Analysis Agency (AFCAA). See Volume 2, Appendix A for information on how to contact the AFCAA.]*

Re-engineering COBOL at WPFA

The **US Strategic Command (USSTRATCOM)**, Strategic Communications Computer Center's, Air Vehicle Application Branch of the **War Planning Systems Directorate's Force Applications Division (WPFA)** is an example of how legacy software can be made more maintainable. They upgraded their A101B-AFAS COBOL modules with automated tools and sound software engineering discipline. Metrics were collected on the existing system (using the SNAPSHOT tool) to discover those system attributes that correlated with the attributes for maintainability (e.g., structure, complexity). The existing code was then **restructured** (using the RECORDER tool). Thorough testing of the modules was performed before restructuring to establish a functional baseline for the existing code. After restructuring, the code was tested again to establish a *post-functional* baseline. The two baselines were then compared to confirm that the functionality of the code remained the same. The WPFA team realized that the key to their success was making the user realize the value of their efforts. They convinced their users that:

- System reliability had increased;
- Future maintainability would improve significantly, thus extending the software's useful life; and,
- The restructured system would remain functionally equivalent.

The software's maintainability attributes were improved from **34** to **98** (maximum of 100). Overall complexity was reduced from **90%** to **9%**. The team did express a word of caution about restructuring legacy software. Even good automated tools cannot turn garbage code into good code. Tools can do an excellent job of restructuring reasonably good code, making it easily readable and more structured. But their ability to improve poorly-written modules is limited. If you are responsible for the maintenance of garbage code, and it can be localized into stand-alone modules, ***re-engineer them to Ada***. The cost to do so will be recouped while avoiding long-term maintenance headaches. [BLUNK92]

STSC Re-engineering Support

The **Software Technology Support Center (STSC)** is available to support your re-engineering efforts. STSC efforts have included:

- **JOVIAL Re-engineering**. A JOVIAL re-engineering toolset.
- **COBOL Re-engineering**. This report includes a list and evaluation of the top COBOL re-engineering tools now available.
- **Software Re-engineering Center (SRC)**. The center performs JOVIAL and/or COBOL re-engineering, tests commercially available re-engineering tools, and trains personnel in the use of re-engineering tools and techniques.

The STSC August 1994 report, *Re-engineering, Volume 1,* is a comprehensive evaluation of all known available tools for re-engineering. It contains a methodology for a quick determination as to whether your software is a viable candidate for re-engineering. It then directs you where to pursue more in-depth models for judging the costs associated with re-engineering. [STSC92] *[Another good reference is McClure, The Three Rs of Software Automation: Re-engineering, Repository, Reusability, 1992.]*

NOTE: See Volume 2, Appendix O, *Additional Volume 1 Addenda*, Chapter 11, Addendum B, *Electronic Combat Model Re-engineering.*

LOGISTICS SUPPORT ANALYSIS (LSA)

It has not been the practice for contractors to perform formal **LSAs** for software acquisitions. Even for weapon systems, most LSA, is confined to hardware. A complete, well-rounded approach to assuring that software is supportable has not been formally developed. In 1991, at the 26th Annual International Logistics Symposium sponsored by the **Society of Logistics Engineers (SOLE)**, a paper was presented by A.G. Johnson and T.A. Haden, from the United Kingdom Ministry of Defense Army Electronics Branch. This paper included a **Software Supportability Checklist**, modeled after those used for hardware. It is reproduced in Table 11-1 (below) for the benefit of program managers and contractors who desire to give additional attention to the LSA of their software.

	SOFTWARE SUPPORTABILITY FACTOR	DESCRIPTION
1	**Maintainability**	Requirement for a Maintenance Task Analysis (MTA)
2	**FTA, FMECA**	Requirement for Fault Tree Analysis (FTA) and Failure Modes and Effects and Criticality Analysis (FMECA) to be performed to functional depth
3	**Defect Rate**	Requirement to state a contractual target defect rate per lines of code over an agreed period including confidence limits
4	**Failure Identification**	Design to provide features that achieve failure detection and location times
5	**Failure Snapshot**	Design to provide features that achieve failure detection and location times
6	**Tool Kit**	Provision of User/Maintainers software tool kits to aid failure location
7	**Loading and Saving Data**	Design to allow loading or saving data in specified times
8	**Configuration Identification**	User/maintainter able to identify the configuration status (version) without accompanying documentation
9	**Exception Handling**	Design to allow exception handling to preclude failure conditions from aborting software during operations
10	**Support Policy Constraints**	Use Study to include what the software must do and not do
11	**Support Maintenance Policy**	Support specific maintenance policies and manpower ceilings and skill level availability to be stated
12	**Software Support and Maintenance Categories**	Categories of software support and maintenance to be stated
13	**Media**	Proposed media must: (a) suit the environmental requirements, and (b) be acceptable as a consumable item
14	**Media Copying**	Simplify copying and distribution
15	**Media Marking**	To allow physical and internal marking; safety critical items to be separately marked
16	**Packaging**	Media packaging to be consumable, reusable, and robust
17	**Handling**	Media to require no special precautions and meet Use Study requirements
18	**Storage**	Media to require no special precautions or facilities and meet Use Study requirements

Table 11-1 Software Supportability Checklist

	SOFTWARE SUPPORTABILITY FACTOR	DESCRIPTION
19	**Transportation**	Media and packaging to require no special requirements
20	**Training, User**	User training required to detect failures and invoke exception handling
21	**Training, Support**	Support training required to detect and locate failures and invoke exception handling
22	**Publications**	User and Support publications will be required
23	**Definitions**	The Requirement must include contractually agreed upon definitions of: incident, fault, failure, defect, reliability, and failure categories
24	**Resources**	Cost estimates must be sought for software maintenance
25	**Test Tools**	Contractor-owned and maintained software test tools and documentation must be provided
26	**Test Tool Access**	Access to test tools to be provided to software support personnel
27	**Incident/Failure Reporting**	Incident and failure reporting to be available

Table 11-1 Software Supportability Checklist (cont.)

LSA on the F-22 Program

From the outset, the **F-22 program** has enhanced and implemented **Integrated Product Development (IPD)** and **Integrated Weapon System Management (IWSM)**. Specifically, the program has always integrated software engineers and logistics personnel throughout all **Integrated Product Teams (IPTs).** In addition, the **Life Cycle Software Support (LCSS)** IPT was created to influence software design for supportability and to build a specification that describes the software support concepts for the life of the weapon system. Personnel from product centers, support centers, customers, and contractors work together on the IPTs. Thus, program decisions related to software development and support are jointly determined. Since each IPT is composed of representatives from all disciplines, life cycle impact is always considered as are plans for future software support. Because a software support facility is still some years away, support decisions are analyzed to determine future impact. LCSS IPT personnel ensure that decision makers are briefed on the consequences of support decisions.

NOTE: See master's thesis, *Guidelines for Ensuring Software Supportability in Systems Developed Under the Integrated Weapon System Management Concept*, by Johndro and Butts, Air Force Institute of Technology, December 1993.

Instead of the traditional LSA process, the approach the LCSS IPT used was a combination of parametric models *[discussed in Chapter 10, Software Tools]*, analogy, expert opinion, and top-down analysis *[all discussed in Chapter 8, Measurement and Metrics]*. By analogy, they compared the overall size of the effort to past fighter aircraft designs. The F-22 will have at least twice as much software on board the aircraft as any fighter currently in DoD. Also by analogy, they initially estimated that the average block change magnitude would be approximately 10% of the total source lines-of-code.

NOTE: See Volume 2, Appendix K, "*LSA for Software—A Practical Approach.*"

The F-22 also employs data tables to implement highly volatile functions and reduce the magnitude of block changes. Key design decisions were made to move potential areas of change out of the source code and into the lookup tables. Potential change areas are now isolated to easily modifiable code blocks instead of locked in algorithms. For example, most **Pilot-Vehicle Interface (PVI)** functions have traditionally been hard-coded into the software, but on the F-22, many of these functions will be implemented using data tables. By expert opinion, the IPT leads in charge of software development estimated that the use of data tables would reduce the block change size by about 50%. Once the overall effort was estimated, parametric analyses of each subsystem provided estimates for schedule and personnel requirements. Three software cost estimation models *[SEER, REVIC, and CostMotio (discussed in Chapter 10, Software Tools)]* indicated varying degrees of schedule and personnel requirements. They then selected a single model to continue a top-down analysis of the large subsystems.

Software support facility cost estimates were also based on expert opinion and analogy. Subject matter experts, such as lab managers and integration and test leads, suggested space and equipment requirements based on F-22 development efforts from which equipment cost estimates were derived. Personnel cost estimates were based on the current annual rates for government and contractor software

development personnel when applied to parametric analysis results. Similar data, which had been previously collected from the F-14, F-15, F-16, and F/A-18 programs, were used for comparison purposes. The comparative data corroborated facility and personnel cost estimates.

An inherent difference between hardware support and software support is that hardware support is based on the finished product, while software support must mimic the development process. Hardware support must use the tools necessary to repair a finished product, not tools required to build a one. Software support, on the other hand, must use tools functionally identical to those used during the development process. To determine F-22 software support requirements, the LCSS IPT started their LSA program by identifying the tools used to create the software. They then developed a software supportability database based on MIL-STD-1388A. Although traditional LSA process was not used to assess software supportability, LSA Record (LSAR) data items are incorporated in a database. Both software maintainers and developers reviewed and commented on the initial database design, as defined by the LCSS IPT. To populate the analysis database, data are collected from the software development IPTs during each development phase. The database is segregated by CSCIs and by development cycle phase. This data collection relationship will continue throughout production and post-production support. The software supportability database implements the intent of the LSA process at the highest level to accommodate software support requirements.

The LCSS IPT will generate several guidance documents for the F-22 program. Specifically, IPT personnel will also prepare and publish a **Post-Deployment Software Support Concept Document (PDSSCD)** as an executive summary of the processes, plans, -and procedures to be used in post-deployment support. System Program Office (SPO) personnel will update the F-22 **Computer Resources Life Cycle Management Plan (CRLCMP)** to reflect software support decisions published in the PDSSCD. Contractor personnel will prepare and deliver a **Computer Resources Integrated Support Document (CRISD)** to define the processes, plans, and procedures for software support. Additionally, contractor personnel will prepare an **Integrated Weapons System Support Facility (IWSSF)** development specification to define and itemize the resources needed to implement the CRISD.

The LCSS IPT takes a very proactive role in the **Software Product Evaluation (SPE)** process. Since the SPE processes keeps software support personnel closely associated with software development teams, support personnel are able to influence design and improve supportability. For example, LCSS IPT and Charles Stark Draper Laboratory personnel developed **Document Evaluation Guidelines** to help evaluate hundreds of software documents generated by the program. These guidelines provided developers with criteria to follow during initial product development. They also form the basis for document SPEs. The LCSS IPT personnel also train government and contractor personnel on the SPE process so that documents are prepared according to the same guidelines against which they will be evaluated. This dramatically improves the first-time approval rate of software documents. *[NOTE: To obtain a copy of the Document Evaluation Guidelines, see Volume 2, Appendices A or B for information on how to contact the STSC.]*

NOTE: See Addendum A of this chapter for a discussion on the Army's Rapid Open Architecture Distribution System (ROADS) program for providing LSA to the Force XXI Digitized Battlefield.

CONTINUOUS ACQUISITION AND LIFE CYCLE SUPPORT (CALS)

CALS is a collection of standards for developing, storing, and communicating product, parts specifications, and other engineering technical information electronically. The purpose of CALS is to get *on-line* engineering data and specifications for high-tech equipment in a DoD-wide database for easy retrieval and updating throughout a weapon system's life. ***All new weapons systems should include a "delivery-in-place" capability***. This is the electronic capability to deliver *on-request* all contractually required information. Although the data resides with the contractor, DoD retains the rights to the data and must be provided access to it on a **fee-for-service** basis.

MANAGING A PDSS PROGRAM

You employ the same tactics for successful management of PDSS as those employed for new-starts and ongoing software developments. The solutions to your PDSS development problems are the also same software engineering practices used throughout other phases of the life cycle. Unfortunately, you are at the mercy of the initial developer who may have burdened your program with problems. Planning and execution of software support must begin during the concept exploration phase and continue until the system is removed from the inventory. The key areas that must be addressed are illustrated in Figure 11-7. These key areas consist of processes, products, and support systems.

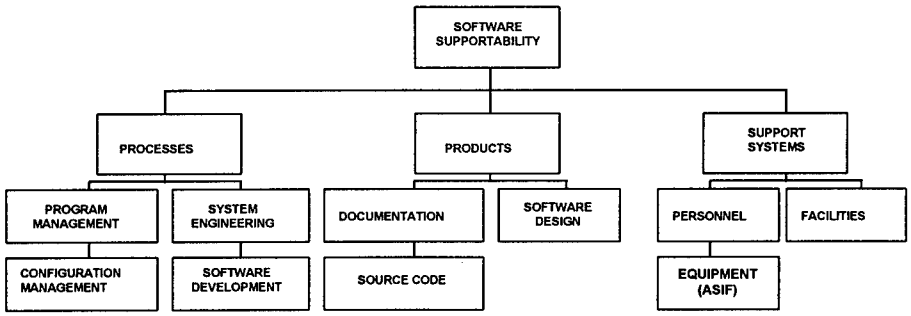


Figure 11-7 Post-Deployment Software Support Key Considerations

NOTE: The 1500th Computer Systems Group Software (CSGS) Development and Maintenance Process Manual, 15 November 1993 (available from Headquarters 1500th CSGS, Scott AFB, Illinois) provides uniform requirements for and guidance on the software development and the maintenance process, and defines specific responsibilities.

Life cycle support strategies typically span the support spectrum from sole source contractor to full government organic, with each strategy presenting different advantages and disadvantages needing evaluation. ***A high level IPT consisting of the operational user, the PEO, and the acquisition agent must make the support decision prior to Milestone I.*** This focuses attention on the software support process and allows the acquisition agent to begin planning for it earlier in the program.

To effectively manage and control software development and to ensure software supportability requires that we incorporate ***measurement*** in the developer's decision making and reporting processes. With measurement, we can monitor the development effort, gain early insight into potential problem areas which can negatively impact the PDSS task, and ease verification procedures. *[See Chapter 8, Measurement and Metrics, for a discussion on measures that track supportability.]*

Support **processes** are the most important element for management, control, and improvement of software support. The key processes that must be captured and recorded are ***program management, configuration management, systems engineering***, and ***software development***. The key **products** essential to PDSS are documentation, source code, and a description of the software design and test procedures. The baseline for PDSS activity is the delivered products from the initial development. The effectiveness of PDSS is governed by the usability and descriptiveness of the delivered documentation. Source documents for these essential products are contract CDRLs, CLINs, and the CRISD. Support **systems** include the people, facilities, tools, and equipment needed to perform the maintenance task.

The following are key management activities to remember for PDSS success:

- Determine your life cycle support strategy early,
- Remember that software support is actually software re-development,
- Ensure adequacy of contractor software development processes during source selection,
- Identify supportability requirements and objectives in systems requirement documents and Statements of Objectives,
- Specify required documentation and verification methods in the appropriate CDRLs,
- Identify necessary software development and support tools in CRISD, and
- Establish a CRWG IPT.

Computer Resources Integrated Support Document (CRISD)

The **CRISD** is the key document for software support. It determines facility requirements, specifies equipment and required support software, and lists personnel number, skills, required training. It contains information crucial to the establishment of the SEE, its functionality, and limitations. It is a management tool that accurately characterizes the SEE's evolution over time.

The CRISD is a product of the software development process. As the pyramid in Figure 11-8 illustrates, the bottom tier, "*early analysis and supportability decisions during design*," is the cornerstone in achieving supportable software. Support requirements and characteristics must be specified at the beginning of the design process so that supportability features are an integral part of system development. This permits identification of support resources as they are needed. It also enables the identification and documentation of the software development tools used. The CRISD is a *living document* that reflects the development configuration and test/integration environments. Thus, the CRISD lays the foundation for PDSS and is essential in reducing software life cycle support costs.

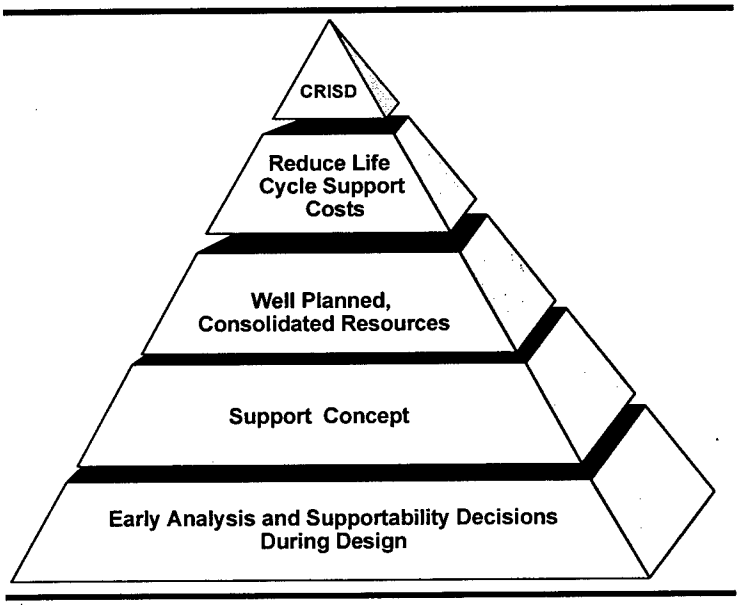


Figure 11-8 Computer Resources Integrated Support Document (CRISD)

NOTE: See Volume 2, Appendix K for a discussion on the importance of the CRISD and how it is being implemented on the F-22 Program.

Society of Automotive Engineers (SAE)

The **SAE Software Subcommittee** has established a program to capture and implement software supportability techniques and methods. The primary thrust of the program has been to apply an **Integrated Logistics Support (ILS)** methodology across the software life cycle with logistics personnel playing a major role in the development process. The SAE software subcommittee products provide a significant educational tool for raising the awareness of software support. The SAE G-11 Reliability, Maintainability, and Supportability (RMS) Software Subcommittee has established the following goals and objectives to address software support.

- Support the development of software RMS standards,
- Produce handbooks for the implementation of software RMS,
- Promote awareness of software RMS among international industries and governments, and
- Promote education and training of engineering professionals on the application of software RMS. [SAE95]

To implement these goals and objectives, the SAE Software Subcommittee has established the following programs:

- Project G-11-SS-95-1: Software Supportability Overview
  - *Handbook for Software Support Application*
- Input for Design Guide for the achievement of Software Reliability (UK MOD Defense Standard 00-42)
  - *Software Reliability for Safety Critical Systems*
- Project G-11-SS-95-3: Logistic Support Analysis Application to Software Aspects of System (UK MOD Defense Standard) 00-60

ADDRESSING SOFTWARE SUPPORT IN THE RFP

Supportability is one of the most important issues to address in the RFP. Your RFP must require that offerors plan for supportability by stipulating that the software be developed with a ***supportable architecture that anticipates change, uses accepted protocols and interfaces, and has documentation consistent with the code***. This can only be achieved during initial software development and must be addressed upfront in the development contract. The higher the quality of the initial system, the easier it will be to support. Therefore, the offeror's approach to supportability must be a major source selection criterion.

In 1990, a survey of over 100 businesses and technical people conducted by the **Air Force Scientific Advisory Board** revealed that contractors do not perceive supportability and maintenance as important factors for winning software development contracts. This study showed that software contractors believe cost, performance, and schedule are the Government's main concern. This perception by contractors must be changed. The primary vehicle to help institute this change, especially for your program, will be the emphasis given to supportability in your RFP. [PDSS90]

One method to emphasize the importance of supportability is to require pre-award competitive software exercises (e.g., prototypes and demonstrations). These compute-offs can be followed by multiple awards for design demonstrations. The design demonstrations are based on evolving, value-added prototypes that ultimately converge into a fully supported product at the end of the initial procurement. To make this acquisition strategy effective, the developing contractor(s) must be forced to support previous, but evolving, versions of the product the same way a PDSS maintainer would. The prototype developers are forced to select design(s) that promulgates a low-cost, efficient solution with minimal side-effects on software maintenance. The subsequent EMD development contract is awarded to the most supportable design.

Whether a contractor maintains the software, or it is transitioned to in-house government maintainers, the maintainer must have the original developer's SEE and other essential tools for proper code maintenance. The following deliverables must be required:

- Data rights to make and install changes,
- Source code and documentation adequate to understand the code,
- Computer resources (SEE, computers, compilers, etc.) needed to modify the source code and produce object code,
- Equipment and support software to test the subject code, to diagnose problems, and to test solutions, enhancements, and modifications,
- Equipment needed to distribute and install the new software,
- A workable system to identify problems, resolve new requirements, and manage the support workload, and
- Skilled personnel to perform required maintenance tasks. [ALC89]

The way you structure the RFP to acquire and develop your initial software can profoundly impact the availability and usefulness of the required support environment. Therefore, ***you must require that all offerors describe their plans for supportability as part of their proposal submission.*** To ensure a prospective offeror's systems engineering and software development processes adequately address the supportability of software, it is imperative you carefully evaluate the offeror's software development processes during source selection. To do so three major areas must be addressed:

- **Software Development Plan (SDP).** *[Discussed in detail in Chapter 14, Managing Software Development.]* Require the submission of a SDP with offerors' proposals that states how the they intend to ensure their development process addresses supportability relative to the systems engineering process. This plan is evaluated during source selection.
- **Capability Evaluation**. *[Discussed in detail in Chapter 7, Software Development Maturity.]*
- **Instructions to Offerors (ITO).** The ITO and source selection evaluation criteria must specifically address those areas you consider critical processes. The evaluation criteria should describe what is required of the offerors' proposal and how it will be evaluated. The Aeronautical Systems Center has developed a RFP template which provides general and specific guidance on preparing the RFP for software-intensive systems. *[You might also consider requiring that offerors address the software supportability instructions contained in Volume 2, Appendix M as part of their proposal response. In addition, Appendix M provides a shopping list of RFP statements, definitions of software supportability metrics, and a sample "Instructions to Offerors" that addresses supportability.]*

Specifying Supportable Software

Acquiring supportable software also requires the specification of software product performance requirements. The major instruments contained within the RFP are illustrated in Figure 11-9.

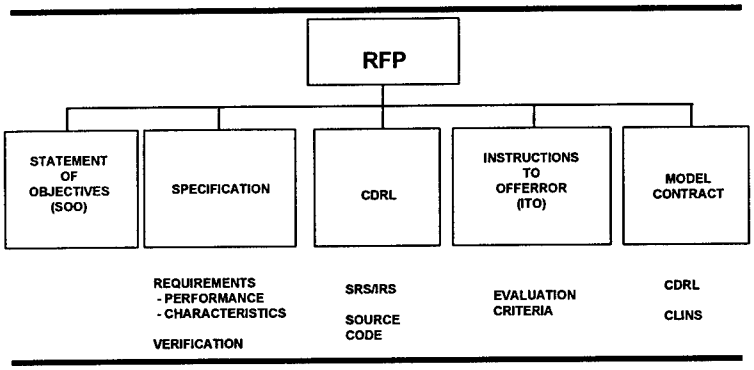


Figure 11-9 Acquisition Instruments

Statement of Objectives (SOO)

The **SOO** defines an objective for efficient, life cycle software support consistent with total system requirements. The SOO states that software supportability requirements and support characteristics are to be managed as an integral part of system development.

Specification Practices

In accordance with the Perry Memo *[discussed in Chapter 2, DoD Software Acquisition Environment],* your RPF must describe *what you want* to procure — not how to design or build it. You can provide top-level system specifications or requirements documents to satisfy the *"what you want."* These specifications can only contain performance requirements and key system characteristics — they can not contain design solutions or detailed design requirements. You can describe the methods you intend to use to verify that system requirements have been achieved. For each performance requirement a corresponding method of verification should be provided. Therefore, specify ***key software supportability characteristics*** along with corresponding verification methods in the system specification or requirements document. As outlined in **Air Force Guide Specification (AFGS) 87253B**, specify the following characteristics to ensure your software acquisition is supportable:

- **Module size**. Module size affects software supportability. Module size [a typical computer software component (CSC)] should generally not exceed 100 SLOC.
- **Complexity**. Application complexity affects software supportability. One generally accepted complexity measure is **McCabe's Cyclomatic Complexity Measure**, which should not exceed 10 for a given module. *[McCabe's complexity measure is discussed in detail in Chapter 8, Measurement and Metrics.]*
- **Programming language**. Software developed in widely-accepted, higher order programming languages enhance software supportability. Ada is the DoD higher order language of choice.
- **Spare memory**. The availability of installed, spare memory software supportability. Spare memory permits the incorporation of enhancements and the correction of latent deficiencies. The effect of spare memory on supportability was calculated for the E-3 AWACS where two similar radars were delivered with 9% spare and 34% spare memory, respectively for the APY-1 and the APY-2. Measurements revealed a 3:1 cost and schedule impact when making the same change to both E-3 radars.
- **Spare computer throughput**. The availability of installed, spare throughput affects the software supportability. Spare throughput permits the incorporation of enhancements and the correction of latent deficiencies.
- **Spare computer system input/output**. The availability of installed, spare input/output affects software supportability.
- **Other parameters.** These include **Halstead Metrics** *[discussed in Chapter 8, Measurement and Metrics]*, SAIC, Inc.'s *Quality Profile Metrics for Supportable Maintainable Software*, the IEEE's software reliability concepts as they may apply to specifying a required level of software supportability, and Rome Laboratory's, Framework Implementation Guidebook, RL-TR-94-146, August 1994.

Documentation

Because software is unlike any other product, the only way to visualize and understand it is through its documentation. *[Documentation is discussed in detail in Chapter 14, Managing Software Development.]* Without accurate, high-quality documentation, software cannot be understood. In essence, documentation is the most important aspect of software support. Documentation delivery requirements specified in CDRLs include:

- Software and Interface Requirements Specifications,
- Software and Interface Design Descriptions,
- Database Descriptions,
- Software Product Specifications,
- Source Code Listings,
- Test plans/descriptions/reports,
- Software Development Plans,
- Software programming manuals,
- Software users manuals, and
- Software maintenance manuals.

The specific criteria for government acceptance of software design information should be clearly specified in the appropriate CDRLs (DD Form 1423) items. This includes the verification methodology, composition of the verification teams, and quantitative thresholds that must be met or exceeded. Offerors should be encouraged to provide alternative verification approaches.

Life Cycle Software Support Strategies

To ensure the contractor's process for developing the software addresses information and documentation management, quality, and verification procedures, typical life cycle support strategies available for source selection include the following.

- **Sole source (original contractor).** The original contractor is awarded the software support contract. The processes, products, and support system are already in place at the contractor's facility and typically are the same as those used during the development.
- **Competitive (support equipment provided).** A competitive contract is awarded and the processes, products, and support systems are either transferred from the original contractor facility to the competing contractor or the items are duplicated. The original contractor can also be a competitor.
- **Organic/contractor mix.** The Government and the contractor share responsibility for software support. Each agent is assigned a percentage of the software to be supported. Typically the Government and contractor are collocated. The processes, products, and support system is relocated to a government support center or the items are duplicated. Manning of the effort is shared by the Government and either the original contractor or a competitive contractor.

- **Organic.** The Government assumes responsibility for software CSCIs. The processes, products, and support systems are relocated to a government support center or duplicated. Support processes are executed by government organic personnel.

REFERENCES

[ALC89] *Supportable Software Acquisition Guide*, First Edition, San Antonio Air Logistics Center, October 1989

[ANSI/IEEE83] ANSI/IEEE Standard 729-1983, IEEE Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers, Inc., New York, 1983

[BASSETT95] Bassett, Paul, "Maintenance is a Misnomer," *Software Magazine*, December 1995

[BISCHOFF91] Bischoff, Col Ron, as quoted in "Design and Planning Make High-Tech F-22 Easy to Maintain and Support," *Aviation Week & Space Technology*, July 15, 1991

[BLUNK92] Blunk, Scott, "Software Maintenance: Avoiding the Crisis," *CrossTalk*, Issue 32, March 1992

[BOEHM81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981

[BORKY90] Borky, Col John M., as quoted in "ATF Avionics Met Dem/Val Goals, Providing Data for Flight Tests," Aviation Week & Space Technology, September 24, 1990

[GLASS92] Glass, Robert L., Building Quality Software, Prentice Hall, Englewood Cliffs, New Jersey, 1992

[JONES91] Jones, Capers, Applied Software Measurement: Assuring Productivity and Quality, McGraw-Hill, Inc., New York, 1991

[MARSHALL47] Marshall, GGEN S.L.A., Men Against Fire, William Morrow & Co., New York, 1947

[OLSEM93] Olsem, Michael R. and Chris Sittenauer, "Terms in Transition: Re-engineering Terminology," *CrossTalk*, Software Technology Support Center, Special Edition, 1993

[PDSS90] "Report of the Ad Hoc Committee on Post-deployment Software Support," US Air Force Scientific Advisory Board, December 1990

[PERRY93] Perry, William E., "Don't Pour Money Down Rat Holes that Infest Your Budget," *Government Computing News*, December 6, 1993

[PIERSALL94] Piersall, COL James, "The Importance of Software Support to Army Readiness," Army Research, Development, and Acquisition Bulletin, January-February 1994

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[REILY92] Reily, Lucy, "Arms Software Hits Flak: GAO Targets Pentagon on Costs and Scheduling," *Washington Technology*, August 27, 1992

[SAE95] "SAE RMS Minutes," January 1995

[SHUMSKAS92] Shumskas, Anthony F., "Software Risk Mitigation," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[SNEED91] Sneed, Harry M., "Economics of Software Re-engineering," Software Maintenance: Research and Practice, Volume 3, John Wiley and Sons, Ltd., 1991

Blank page.

CHAPTER 11
Addendum A

ROADS: The "Software Logistics Vehicle" for the Digitized Battlefield

Joseph L. Potoczniak
US Army CECOM

Critical and essential to Force XXI is the element of rapid software logistical support. The logistics of software is more rapid than that of hardware logistics and uses different channels that are not encased in today's hardware support or hardware logistics. Software logistical support has been, and continues to be, provided by *ad hoc*, system-by-system, person-to-person, volunteer-by-volunteer processes. Desert Storm, out of necessity, made use of this non-institutionalized software logistical support. What is clear from those lessons-learned is that software logistics needs to be organized and institutionalized to better serve the soldiers in Force XXI.

THE DIGITIZED BATTLEFIELD IS THE SOFTWARE BATTLEFIELD

The phrase "*digitized battlefield*" is another way of describing the software battlefield. Digitized data can only be initiated with software and processed with software and, by its very nature, is software. Documents that are digitized look like software, act like software, behave like software, and are managed like software, thereby showing them to be software. The concept that software is only the lines-of-code that make the systems operate does not recognize that when

voice, intelligence signals, documents, and other means of gathering, holding, communicating, and processing information are digitized, they immediately become the "*ones and zeros*" that flash across memories, disks, cables, and satellites in the manner that any other "ones and zeros", code or not, move. The digitized battlefield is the software battlefield, and the logistics that provide and move the "*ones and zeros*" to where they are needed, when they are needed, is the realm of software logistics. The vehicle of that realm, the very capability that enables it to function rapidly, accurately, and in response to battlefield requirements, is the **Rapid Open Architecture Distribution System (ROADS).**

Why ROADS is Needed: A Case In Point

One story illustrates the difficulties involved with changing software to meet changing battlefield requirements, and why the Army of the 21st century needs to update its capability. It illustrates the need for the incorporation of modern software logistics practices and capabilities to be brought to the aid of the battlefield commander. During Desert Storm, a commander required one of his systems' software to be upgraded as quickly as possible to counter new threats that were encountered. The software that needed to be changed was housed on a circuit card assembly on a semiconductor device.

The supply personnel sent two circuit card assemblies that contained the software back to the Continental United States (CONUS) to have them updated with the needed software changes. This approach was taken because there is little to no institutionalized capability on the battlefield to aid in this particular software logistics process. This actual case, like all other cases of software logistics, is ad hoc, or, in other words, "*get it done the fastest and best way possible.*" In two days, it was discovered that the circuit card assemblies were lost in "normal channels." Additional circuit cards were obtained and they were packaged, addressed, taken to a transport leaving for CONUS, and placed on the transport. The tail number of the plane was recorded by the supply personnel and then telephoned ahead to McGuire Air Force Base (AFB), where another team would be waiting for the plane with the specified tail number.

Once the plane was observed, it was greeted by the waiting supply personnel, and the package was removed and taken to Fort Monmouth to the basement of the CECOM RDEC SED Building, where some engineers spent the night updating the circuit cards, packaging

them, taking them back to McGuire AFB, and placing them on a transport headed back to Desert Storm. With the tail number of the transport recorded, they telephoned the number ahead to the waiting supply personnel in Desert Storm, who intercepted the aircraft and obtained the package with the circuit cards in it. The circuit cards were then installed in the system and the job was done. Thus ended one cycle of a software logistics exercise.

The Problem That is Illustrated

The example serves to illustrate the problem of how software logistics functions today: essentially by the seat of the pants (more generously and politely called "*ad hoc*"). This sample software logistics exercise is multiplied by the hundreds of battlefield systems that need software updates, then the numbers of personnel, the specialized handling, the time consumed, the battlefield opportunities lost due to not having the updated software, the missions missed, and the potential cost of lives, just by not having the software rapidly updated to meet the commander's requirements for his battlefield situation.

Current practices have no institutionalized guidance, and each battlefield system's managers use their own judgment or local organizational practices to effect a solution to the anticipated problems that will be encountered during the life of each system. With software, there is no institutionalized list of lessons-learned, and even local practices allow for ad hoc and individual approaches. There are no present Army processes that could be provided for the lessons-learned for each manager. The equipment, tools, processes, organizational responsibilities, staffing, and doctrine are not in place. The field of software logistics is newly emerging and is not even recognized in some circles. The difficulties arising from the Army software logistics vacuum are sometimes seen as system difficulties, not software logistics issues, thereby obfuscating the vision of the solution or even the fact that software is involved with the problem and its solution.

While many issues can be discussed, one issue is most critical: what is the best way to support software for the present and 21st century battlefield commanders on the digitized battlefield. The Army of the future and its abilities to fight will be dangerously anchored to a host of inadequate past practices unless the backbone of software logistics is recognized and institutionalized. That backbone is called ROADS, which is the acronym for Rapid Open Architecture Distribution

System. The need for this system has been illustrated by the example presented above and the discussion that surrounds it. The commander will have to be aware of software and its impact on his abilities as with any other capability at his disposal, possibly more so. The point to be emphasized is the fact that the flood of battlefield software is rising with the materiel release of each system and the rising production of systems on the production lines. These systems are software-intensive, in that they depend largely upon software for the abilities they provide the fighting soldier.

ROADS

ROADS is a mobile, sheltered vehicle that has the capability to receive software from a variety of sources, such as satellite, fiber-optic cables, copper cables, or tactical systems. ROADS then places the software on the necessary medium, such as firmware, floppy disks, tapes, or CD-ROMs. Maps will be especially important. Placed on the battlefield, ROADS provides the backbone for updating the commander's software in battlefield systems.

ROADS, in its present concept, will be configured from an M1097 High-Mobility Multipurpose Wheeled Vehicle (HMMWV), complete with a shelter containing replication equipment, communications capabilities to help coordinate and distribute the software, and a trailer in tow that will house a satellite communication capability, cables, and necessary antennas (see Figure 11-10).

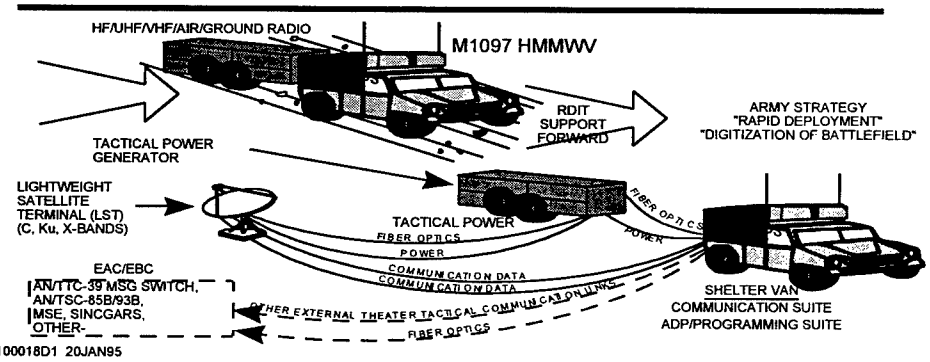
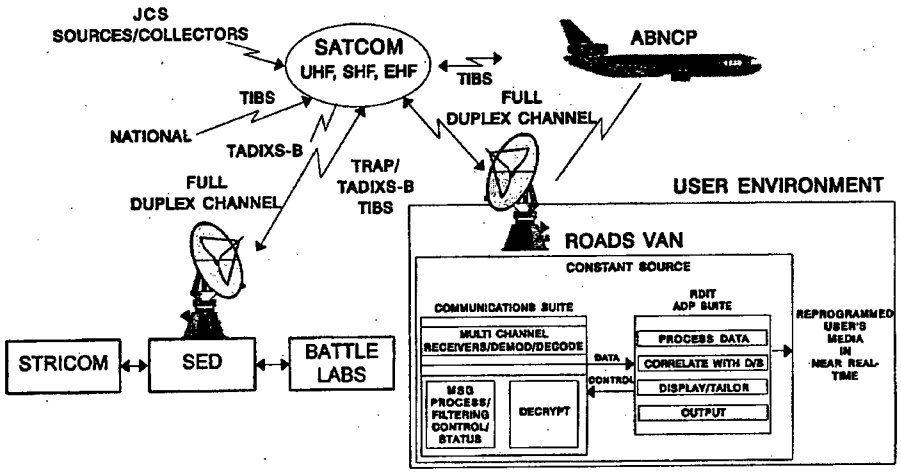


Figure 11-10 Prototype ROADS Model Configuration

It is critical to the battlefield mission software readiness and sustainability support capabilities of ROADS that it be capable of several alternative means of receiving and sending software, coordinating its distribution, and managing the overall process. Toward this end, several bandwidths must be available.

ROADS Doctrine

Doctrine for the deployment, staffing, use, organizational responsibilities, and deployment process needs to be defined. Whether the system will be assigned to each of the Army's divisions, at echelons above corps, or some combination of both is yet to be determined. The overall concept of how ROADS will function on the battlefield is shown in Figure 11-11.



TIBS - TACTICAL INFORMATION BROADCAST SERVICE
TRAP - TACTICAL RECEIVE EQUIPMENT AND RELATED APPLICATIONS
TADIXS-B - TACTICAL DATA INFORMATION EXCHANGE SYSTEM-BROADCAST

Figure 11-11 ROADS Concept of Operation Constant Source System Architecture

ROADS' role in the Joint and Allied areas is illustrated in Figure 11-12 (below). In this capacity, the concept provides for newly updated software to be transmitted to the battlefield, placed on the appropriate media, and then provided to participating Joint and Allied systems, allowing them the rapid reaction capability needed to interoperate with each other and provide the changes needed to meet the commander's requirements.

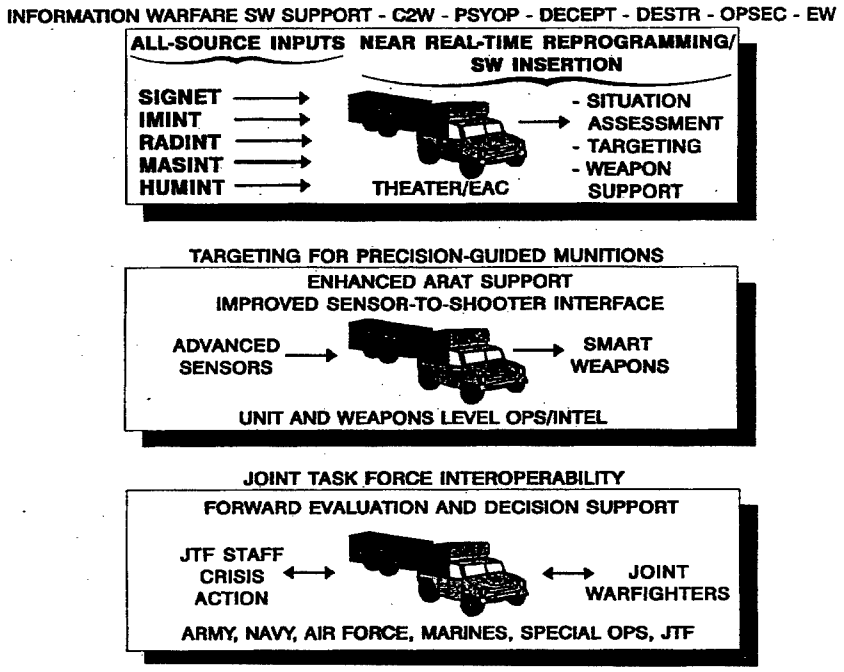


Figure 11-12 ROADS Deployment Examples

ROADS Ancillary Functions

In addition to providing new versions and revised software to the battlefield, there are other critical functions that are provided by ROADS. These functions include labeling the media with the correct software identification, configuration-managing the releases to avoid confusion, standardizing the processes involved, with in-theater software RDIT, and coordinating with the soldier on the battlefield to keep him informed of software issues that may be vital to his interests and requirements.

CONCLUSIONS

Aside from there being no other rapid response software logistics support concept for Force XXI, the benefits derived from ROADS are improved readiness, reduced fielding and post-deployment software support (PDSS) costs, rapid provisioning of software, simplified software logistics, and a standardized approach for software sustainment and supportability on the battlefield. The benefit to the

soldier is that there will be an institutionalized way of dealing with software on the battlefield. One software face to the soldier will be integrated with his other support to provide one face to the field.

The Army is planning for the 21st Century, which is just a few years away. The plans to create the most advanced fighting force in the world should include the plans for the most advanced software logistics support system in the world. To have less may force the Army to be less than it should be on the battlefield.

CHAPTER 11
Addendum B

Electronic Combat Model Re-engineering

Idaho National Engineering Laboratory
March 1995

NOTE: See Volume 2, Appendix O, *Additional Volume 1 Addenda* for this article.

PART III
Management

Blank page.

CHAPTER

12

Strategic Planning: The Ounce of Prevention

CHAPTER OVERVIEW

In this chapter you will learn that ***planning*** *is your first and most critical task as a manager. Program stability, quality, on-time completion, within budget, sufficient performance, and efficient supportability are all important strategic planning goals. Be aware, however, there is no set formula for determining an acquisition strategy that will achieve these goals. Every program has its individual planning needs which must be reflected in your acquisition strategy.*

System and software acquisition objectives and scope, required resources, management assumptions, extent of competition, proposed contract types, program structure, and, most importantly, program risk strategy are all included in your acquisition strategy. A good acquisition strategy is the foundation for subsequent contracting and budgeting activities and is based on three key premises: (1) you must start full-scale development with a good software acquisition contract; (2) you must inspect what you expect through close monitoring of contract requirement compliance; and (3) you must make sure the software is supportable after system deployment. [MICOM91]

Definition of development objectives and scope, decomposing the program into a work breakdown structure (WBS), an System Segment (Subsystem) Specification (SSS), manageable tasks, and estimating software size, cost, resources, and schedule are all critical program planning activities. There are many methods and techniques for producing these estimates, none of which are 100% accurate. The best approach is to employ several methods which can be cross-checked against each other to arrive at a composite estimate. Remember, all your planning efforts are a waste of time unless you actually implement your plan and re-implement your planning process as the program environment evolves over time. All initial estimates must be updated with actual program data to consistently refine your cost, schedule, and resource projections.

Your acquisition strategy must be tailored to your program's needs and the amount of risk you are willing to accept. With software, flexibility and accommodation for evolutionary changes are important factors to consider

when developing your strategy. You must also strive for realism, stability, resource balance, and controlled risk [see Chapter 6, Risk Management]. The basic elements of your acquisition strategy must be predicated on a thorough and knowledgeable investigation of alternative solutions. Remember, you can reduce and/or eliminate major software program risks if you apply sound software engineering practices based on realistic estimates of software cost and delivery schedules.

Schedule changes and slips are another risk area of major concern. Not only do they affect your effort, but the schedule of the overall system when the development of other system component must wait for a late software delivery. Not meeting performance requirements can often be traced back to improper dedication of resources to the requirements analysis and design phases. The best way to mitigate these planning risks is having a skilled team that performs careful upfront cost, schedule, and performance estimates, and continuous program tracking and control based on actual metrics data collection and analysis. ***People*** *are your greatest resource. Employ the highest skilled, most experienced people available.*

CHAPTER
12
Strategic Planning: The Ounce of Prevention

PLANNING IS KEY TO SUCCESS

An analogy can be made between planning for the acquisition and management of major software-intensive systems and planning for a military campaign. Planning for combat was explained by **General H. Norman Schwarzkopf**, former Commander US Central Command, as:

> *I want to emphasize the importance of focusing on the enemy when planning and conducting combat operations. First, you must know your enemy. Second, you must develop your plan keeping the enemy foremost in mind. Third, you must wargame your plan to enhance your ability to develop or adjust the plan once enemy contact is made.* [SCHWARZKOPF88]

You must view cost overruns, schedule slips, and performance shortfalls as your enemy. You must also emphasize the importance of focusing on risk when planning for and conducting your program. First, you must know your program-specific risks. Second, you must formulate your strategy to enhance your ability to develop or adjust your plan as you encounter new sources of risk in the ever-changing program environment.

World-class software doesn't just happen — it's planned! **Planning** is the most pivotal activity you will perform as a program manager. Planning, combined with process improvement, is a continuous activity that must be revisited and improved upon

throughout the life of a software-intensive system. A poorly planned software program is one that is doomed to failure. Through proper and careful planning you can address and deal with the five critical factors that determine the success or failure of a software program: quality, cost, schedule, performance, and supportability. Although software planning is performed throughout the software life cycle, ***upfront, strategic planning*** is the most crucial. It addresses these critical planning factors that get exponentially more costly to deal with in later phases. **Software development** is not an exact science, but using a combination of good historical data and systematic techniques can improve the accuracy of your estimations. The **F-22** program illustrates that software development success is achievable through careful risk management and knowledgeable strategic program planning that combines a keen sense of lessons-learned with a commitment to achieve insightful, intelligent, and creative process improvement. It involves interaction with Air Force agencies and strategic planning stakeholders to arrive at the best software solution within budgeted resources.

In April of 1991, **General Merrill A. McPeak**, Air Force Chief of Staff, was proud to announce the winner of the advanced tactical fighter (ATF) air superiority aircraft competition and the Air Force's new *Top Gun*. The contract award for the future, fast, agile, stealthy super cruiser, the F-22, was the result of a 54-month Demonstration/ Validation (Dem/Val), where two contractor teams dueled for Air Force favor in an unprecedented, risk reducing, joint government/ industry-sponsored face-off. [EASTERBROOK92] Applying lessons-learned from the **B-2**, Air Force planners had the more complex ATF fighter airborne in just four years. For the F-22 software development effort, strategic planning made the difference. Our ATF planners must be commended on their success. Former Secretary of the Air Force, **Donald B. Rice**, praised them when addressing the House Armed Services Committee after the award of the F-22 contract. He stated that there has never been a defense program "*that invests as much in the front end...and is in as confident a position to enter full-scale development as the ATF.*" [RICE91] The critical nature of software in the weapons and information systems you are developing or maintaining today, mandates that you take every action necessary to ensure program success.

When trying to assure the overall success of your program, there are three important points to remember about software. First, usually the biggest cost item in major DoD software-intensive acquisitions,

software is always on the critical path. Software, so vital to military systems, is also the ***highest risk item that must be steadfastly managed***. Our ATF planners used a strategy that included lessons-learned from the **C-17** development where the software element was unrealistically considered a low-risk item. [REILY92] To better manage the F-22 software development, they decided that software costs must be tracked separately from hardware costs in the Engineering Manufacturing Development (EMD) phase. This is the first time the Air Force has taken this approach to manage and reduce the risk of software cost escalations on a new aircraft. [HUGHES92]

The second point to remember when planning your software acquisition is consistency and completeness. **Consistency** means having single standard languages (i.e., Ada) where possible, a standard terminology, a standard software engineering environment (SEE) used by all subcontractor team members, and a strong configuration management program. **Completeness** means, quite simply, ***good documentation*** *[discussed in Chapter 14, Managing Software Development].*

A WORD OF CAUTION: In planning for "*good documentation*," do not fall into the "*excessive*" documentation trap! Less required documentation (i.e., only that which is necessary for technical software development and maintenance) of higher quality is the goal.

The third point to remember is to ***keep government personnel*** abreast of the development process so they know what is going on and understand the software and how it works. An important ingredient in program success is to be an ***enlightened*** and ***supportive*** customer. If the contractor encounters unforeseen problems, do not criticize and throw rocks — but cooperate in the search for a solution.

STRATEGIC PLANNING GOALS

At the Pentagon there is a sign posted above the DoD Joint Staff office door (quoted from Field Marshal Helmuth Graf von Moltke) that says:

> *Planning is everything. Plans are nothing.*

In Chapter 4, *Engineering Software-Intensive Systems*, you were introduced to the concept of a **process-focused approach** to problem solving, as opposed to a *product-focused approach.* This is the meaning behind von Moltke's statement. Planning success is achieved though the **planning process**.

In 1988-89, the **Defense Systems Management College (DSMC)** and Harvard University analyzed the findings (in addition to extended research of their own) of several commissions held over a twelve year period to determine why there were so many defense acquisition program failures. The objective of the study was to determine how Government might learn to ***"do business like business."*** [DSMC89] This theme was reiterated recently by **John Deutch**, Under Secretary of Defense for acquisition and technology when he said:

> *We have built up a separate way of doing business with DoD that is entirely different than... the commercial sector. There will be a time when a separate defense industrial complex will become too costly to maintain. We have to learn to rely much more strongly on doing business like the private sector.* [DEUTCH93]

The measures of software-intensive program success are as follows and are as illustrated in Figure 12-1.

- **Program stability**,
- **Quality** (including performance) *[how to measure quality is discussed in Chapter 8, Measurement and Metrics; how to build quality in is discussed in Chapter 14, Managing Software Development; and how to improve quality is discussed in Chapter 14, Managing Process Improvement]*,
- **Supportability** *[how to measure supportability is discussed in Chapter 8, Measurement and Metrics; Software support is discussed in Chapter 11]*, and
- **Cost and schedule** *[how to measure and estimate cost and schedule are discussed in Chapter 8, Measurement and Metrics].*

CHAPTER 12 Strategic Planning

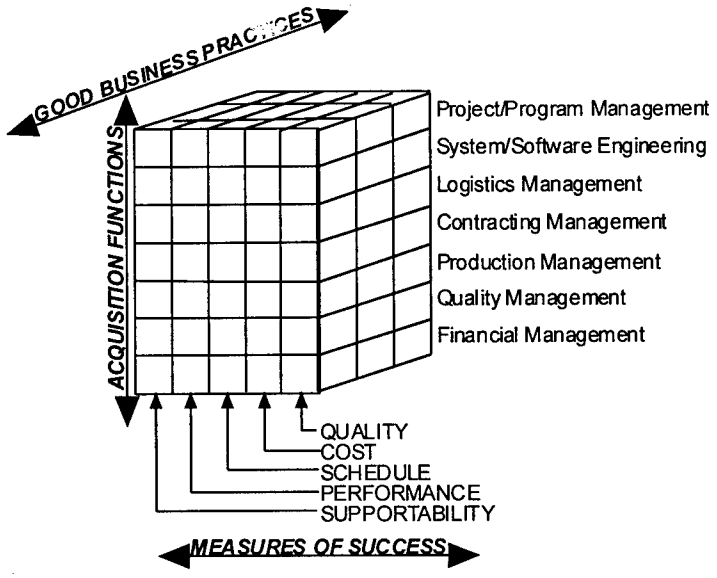


Figure 12-1 Factors Influencing Program Stability

Achieving stability, quality, supportability, schedule, and budget are all determinants of software program success. Luckily, trying to achieve the first three goals helps in achieving the others, and vice versa. A realistic cost and schedule that enables built-in product quality enforces on-time completion which enhances program stability. Program stability, cost, and schedule are always joined at the hip. A stable program can be executed more quickly than one which is constantly changing or subject to change in an unforeseen way. Also, a program completed within its projected schedule is subject to the forces of change for the minimum time possible. Poor quality can cause schedule overruns leading to cost overruns. This can be particularly true where the true quality is not known until final testing is complete and/or the software is in the user's hands. Post-deployment might be the first time you learn that your software simply does not work. ***Lessons-learned from past DoD failures to meet schedule objectives show that schedule slips promote excessive changes in requirements.*** Users having to wait inordinately long periods before their needs are satisfied, invariably identify additional requirements as time and technology advance. The further out requirements must be projected due to schedule slips, the more technologically impractical they become.

Program Stability

> *Recognition of the distinction between a stable system and an unstable one is vital for management. The responsibility for improvement of a stable system rests totally on the management. A stable system is one whose performance is predictable. It is reached by removal, one by one, of special causes of trouble, best detected by statistical signal.* — W. Edwards Deming [DEMING86]

Granted, the DoD acquisition process has environmental factors not found in the commercial world, such as Congressional oversight with one year funding constraints on how and where Defense dollars are spent. Given these differences, it is uniformly acknowledged that **program stability** is the one business practice that should be institutionalized in DoD acquisition policy. It is also the primary goal you should strive to achieve in your program planning efforts, which should ripple across all traditional acquisition functions (e.g., engineering, logistics, and financial management). The key attributes of program stability are ***steadiness of purpose***, a ***firmly established plan***, and a ***supportive system***. Your strategic planning process must link program objectives to resources (time, people, funds, and technology). *[Resource estimation is discussed in Chapter 8, Measurement and Metrics.]* It must organize these resources and define a process for achieving the approval of all stakeholders to guarantee the implementation of your strategic plan. It must then guide the development phase and provide for the integration of the effort. Your approved plan should be a product of systematic consensus and a clear decision process.

Maintaining stability in a program that must be accelerated in order to meet a military threat, or that has had its budget cut by 50%, is often a significant management challenge. However, the steps you take to achieve stability (i.e., having an established plan and understanding how your resources are tied to program objectives) can help you restabilize after a change. Not having a flexible plan that adapts to change will often mean chaos added to chaos when your budget gets cut or requirements are added or modified. The time you spend defining your acquisition strategy early on will go a long way in assuring stability throughout the entire life of the system, especially during the critical acquisition years. The **Cost Analysis Requirements Document (CARD)** *[discussed below]* is an excellent tool for structuring your program for stability.

Quality

Without exception, the second most important goal must be product **quality**. In the past, the goals of cost and schedule often took priority over quality because they had the highest visibility during early phases of development, because until the testing phase, quality is essentially an *unknown* or *invisible commodity*. The rewards for software with good operations and maintenance records are usually not enjoyed by the program manager who developed and delivered the product. This lack of positive feedback creates managers who let costs and schedules drive their decisions — often at the expense of quality. This is "*penny-wise and pound foolish!*"

If you let cost and schedule take priority over quality, by the time your software is coded and ready for testing, it can be so riddled with defects that dynamic testing is painfully time-consuming, costly, and difficult. Once the software is in the user's hands, poor quality becomes excruciatingly visible because the cost to fix garbage code is exponentially greater than the cost of building a quality product (not to mention the damage to user confidence poor quality causes). [KINDL92] Look again at the "*traditional*" line in Figure 10-2. Do not let this happen to you! With quality as a key planning goal, you will produce a ***good product*** on a ***predictable schedule*** at a ***predictable cost*** with the ***desired performance***. Remember, one of the most important ingredients in producing quality software is the belief in the importance of its mission and an associated *commitment* at all levels to support that belief. ***Remember, the true success of your software can only be determined from a life cycle perspective.*** To deliver a quality product you must be willing to adjust cost, schedule, and resource allocation to support the quality goal, however you define that goal.

You must also assure that your contractor employs a methodology intended to promote quality, such as the Cleanroom engineering process *[see Chapter 15, Managing Process Improvement]*. One element of the Cleanroom process is frequent software builds by someone other than the programmer. In the book by Cusumano and Selby, Microsoft Secrets, it is noted that Microsoft uses a similar disciplined "*daily build process*." [CUSUMANO95] In other words, the product is developed by teams working in parallel, but there is "*synch up*" and "*debugging*" ***daily***. This process assures that quality is examined and measured daily. This greatly reduces the opportunity for surprises late in the development and delivery cycle.

Sufficient performance, a quality attribute, is defined in terms of mission capability, supportability, life cycle costs, and unit costs. Beware, rigid or excessive system performance requirements can drive costs unnecessarily high and stretch out schedules. The metrics you use to define your performance goals will ultimately help to determine quality and cost. *[See Chapter 8, Measurement and Metrics, for a discussion on these metrics.]* **Evolving performance goals** should be used, along with **contingency planning,** to facilitate performance tradeoffs should the schedule be jeopardized or development costs become excessive. **Pre-planned product improvement (P³I)** and **evolutionary development** are the standard approaches used to obtain desired technology or features not available at planned schedule cutoff points and milestone decisions. An iterative, evolutionary design process allows for flexible development that advantageously considers performance tradeoffs as the design evolves.

On-Time Completion/Within Budget

The successful F-22 acquisition strategy set a precedent for holding down the price of future DoD weapons systems. Because contractors were required to gamble their own funds, there was great incentive to propose cost-effective solutions. One competing program manager remarked, "*Except for the investment, Dem/Val was great. We probably developed the technology in half the time we would have if we had not had a competition and a good, big team.*" [ROONEY90]

Throughout this chapter the words "***cost***" and "***schedule***" are used over and over because they are two critical metrics used to assess program performance. In DoD, they are often ***the drivers*** used to define program management practices. Their importance is not surprising given the pressing need to update old systems and develop new ones to meet new requirements or threats in today's resource-constrained Defense environment. As **General John W. Vessey Jr.**, while Chairman of the Joint Chiefs of Staff, explained,

> *"Resource-constrained" environment [are] fancy Pentagon words that mean there isn't enough money to go around.* [VESSEY84]

Staying within budget has been one of the most difficult software management goals to achieve. The difficulty arises from the **DoD**

budgeting and **funding process**. Long lead times are needed to get money committed; therefore, program costs must be projected long before software requirements are defined and software cost elements can be realistically estimated. Unfortunately, systems planners are impacted when insufficient dollars are allocated to the software element (on the system's critical path) which in turn often causes the overall program schedule to slip. [MARCINIAK90]

The RFP should require that offerors provide a **development schedule** showing all major milestones, audits, reviews, inspections, and deliverables. You must evaluate this schedule to determine if the offeror understands the need for presenting detailed schedule information and for tying that information to detailed program task requirements. You must also determine whether the program tracking system being proposed is part of the company's normal management practices or if it is new for this program. Also, you will want to ensure schedule needs and types are described and included in the **Software Development Plan (SDP)** *[discussed in Chapter 14, Managing Software Development].*

Problems are often created when schedule baselines are established before software requirements are well defined and understood. Government RFP preparers may include schedule information based on factors that do not take into account the system development process or software requirements. Offerors then inadvertently accept RFP schedule information as a requirement for a responsive proposal, and prepare their response based on these *so-called* requirements. This practice causes offerors to bid to untenable schedules affecting the viability of their submissions, decreasing the probability they will complete tasks as proposed. One solution is to provide minimum schedule guidance, and to require that offerors propose development schedules based on program requirements and their own development approach.

Where users remain adamant that arbitrary delivery dates must be met, you will do well to work with them on the concept of evolutionary and/or incremental deliveries vice a full scope capability. Even then, it is recommended that you use every persuasive power at your command to educate them on the exceedingly high failure rate for programs with unrealistic schedules. *[See Chapter 6, "Software Schedule Estimating," for a discussion on cost and schedule relationships.]*

NOTE: DoD is more interested in receiving a quality product on a predictable schedule at a predictable cost than in setting arbitrary target dates which may not be achievable.

SOFTWARE ACQUISITION STRATEGY

The **acquisition strategy** has been defined as a *master plan, a road map, a blue print, a plan-to-plan-by* to achieve program goals and objectives and to minimize risk. Vladimir Ilych Lenin, master conspirator, strategist, and organizer, victoriously led the Bolsheviks during the Russian Revolution. He claimed that:

> *Any engagement includes the abstract possibility of defeat and there is no other means for reducing the possibility than the organized preparation of the engagement.* [LENIN86]

As you learned in Chapter 6, *Risk Management*, your **Acquisition Plan** must address, deal with, and identify risk issues and alternative solutions. You must decide on what type of contracting strategy to employ, such as design-to-cost, award fee/incentives, or to make-or-buy your software element, which brings with it the issue of data rights. Your development methodology might include concurrency or time phasing of development phases, prototyping, P3I, evolutionary acquisition, and/or incremental development. The supportability of your software must also be part of your acquisition strategy which includes the requirement for an open systems architecture. Other alternatives include: the use and design of reusable assets; re-engineering as a re-development alternative; assessing your potential suppliers' development maturity; tracking and controlling cost, schedule, quality, and other risk elements; and, updating your cost and schedule estimates. Other factors you should address are the cost of scrap and rework, program budgeting and funding risks, a forecast of how future technologies might impact your development, and what kind of planning and management tools you can employ to facilitate your planning activities.

Every major software-intensive development has the abstract possibility of failure. Your acquisition strategy serves as a means for reducing the odds of program defeat through the organized preparation of a plan to minimize software risk. It serves as a guide to direct and control the program, and as a framework to integrate those functional activities essential to fielding a totally operational system — not just

pieces of hardware and software. The conceptual basis of the overall plan — the ***objective*** — is what you must follow during program execution. It also serves as the basis for all program management documents, such as the **Acquisition Plan**, the **Test and Evaluation Master Plan (TEMP)**, the **Integrated Logistics Support Plan**, and the **Systems Engineering Master Plan (SEMP)**.

The critical period for acquisition strategy formulation is during the **Concept Exploration Phase**, when the program manager should be on board. The acquisition strategy is your opportunity to put your personal stamp on the program by meshing your vision for the program with Air Force requirements and the availability of resources. The acquisition strategy is summarized in the **System Concept Paper (SCP)** and other documentation approved through the Milestone I decision process.

There is no common working definition for a standard **acquisition strategy,** no consistent agreement on its structure or composition, nor comprehensive guidance on how to develop and execute it. [DSMC84] Because acquisition strategies for software systems abound, you should conduct a meaningful lessons-learned exercise before settling on your final acquisition and development plan of action. The ATF EMD approach and contract were strongly influenced by careful review of how a whole procession of prior avionics developments went astray. According to **Colonel Borky**, ATF planners based their final acquisition strategy on avoiding the following list of *classic* mistakes that get made over and over.

- *"Unrealistic estimates of time, costs and manpower requirements to execute a development. (Admittedly, we still lack good estimating methods and tools, but there's ample evidence of program managers who willfully understate resource requirements for software to fit within a program budget when they cannot cheat on the hardware estimates.)*
- *Inadequate planning for software integration and test, including required facilities.*
- *Allowing contractors to do significant coding before system engineering is complete and requirements are stable."*

[BORKY91]

In software-intensive systems, software development is considerably more involved than hardware development. Selecting the right strategy may be dependent on technical considerations, such as the complexity of the software integration. An example was the **Federal**

Aviation Administration's (FAA's) Advanced Automation System (AAS) *[now canceled]* that was supposed to further automate the Nation's air traffic control facilities. This was a "*logistician's nightmare*" because the new system was to be installed, tested, and operated in parallel with existing systems until eventually phased-in. Air traffic operations could not stop for the change over — there was to be a transparent transition from the old to the new. [NORDWALL93]

Selecting the right type strategy for your program is much the same as selecting the right strategy for winning a battle. When preparing a battle plan, you must first know what you are getting into and what you have to take with you. Sun Tzu, master Chinese strategist and general during the Era of the Waring States (circa 500BC), explained:

> *Know the enemy and know yourself; in a hundred battles you will never be in peril... If ignorant both of your enemy and of yourself, you are certain in every battle to be in peril.* [SUN500BC]

Achieving each of the strategic planning goals discussed above is a battle in your war to deliver a successful software system. Knowing your enemy and yourself, will enable you to understand your mission, identify your enemy, assess the terrain over which you must pass, and from this, determine your tactics. Your tactics, when combined, make up your overall acquisition strategy.

Mission Definition

Your mission is to deliver a software system that fulfills user requirements, on time, within budget. In some cases, however, the user may not understand exactly what those requirements are, so clarification of the mission may fall on your shoulders. This mission definition includes **timing** (schedule) and **cost constraints** that may have an effect on perceived requirements. The enemy can be equated to all those risk factors that conspire against the completion of your mission.

The enemy to a successful acquisition is not always as clearly defined as is the enemy in a conflict between nations. The enemy may include a variety of programmatic risks or constraints such as: a short schedule or limited budget; factors in the development environment (software being developed concurrently with hardware, or software pushing the leading edge); or, the requirement to build software

designed for reuse and easy maintenance. Assessing the terrain involves analyzing the business base, the capabilities of the development team, and development and host system hardware for which the software is being built.

Acquisition Strategy Development

When preparing your acquisition strategy, you must consider ***all*** the factors pertaining to the requirement such as: budget, technical aspects of the software and hardware on which the software will operate, obtainability of existing software to satisfy the requirement, availability and past performance of organic or contractor developers, timing of hardware and software development, size and functions necessary in the program office, software reusability and maintainability, *ad infinitum*. Obviously the list of issues that can affect the software development strategy is long and complicated and will differ among programs. The bottom line is, you must consider every possibility when selecting contractual, schedule, and budgeting tactics in developing your acquisition strategy. It must be derived from a commitment to encircle, outflank, out think, and triumph over the enemy at every encounter. Your strategy must not be a rigid formula, but a flexible framework that can be applied artfully as circumstances dictate. It must also be based on the application of ***common sense*** and ***best practices*** found in these Guidelines to address the inevitable problems that emerge in every major software acquisition due to the extreme complexity of the endeavor itself. Sun Tzu expressed this strategy when he proclaimed:

> *When the enemy is at ease, be able to weary him; when well fed, to starve him; when at rest, to make him move. Appear at places at which he must hasten; move swiftly where he does not expect you.* [SUN500BC]

If your software requirements are relatively risk free, you might choose as straightforward an approach as possible. If on the other hand, your software is complex and deeply embedded in a hardware system and/or unprecedented, then your acquisition strategy will be quite complex and must involve extensive research into alternatives. For example, avionics software acquisition usually follows airframe development and avionics hardware selection. The airframe developer may then choose to subcontract the development of the very sophisticated software suite needed to mechanize the complex of weapons delivery systems. Planning for this type arrangement will

greatly impact your acquisition strategy. The acquisition strategies (or combinations thereof) commonly used for major DoD software-intensive acquisitions include the following concepts:

- Competition,
- Concurrency/time phasing,
- Design-to-cost,
- Performance demonstrations,
- Performance incentives,
- Make-or-buy, and
- Pre-planned product Improvement (P3I).

Competition

The software industry supplies technology for software development; industry also supplies engineering, development planning, management, organization, infrastructure, and process. These elements were created by industry through efforts to capture market share and gain a competitive edge. Although academia helps sow the seeds of research and training, industry competes to apply better ideas to the software engineering task. Competition is vital to enable technology improvement in defense software acquisition. The forces of innovation are unleashed when industry is challenged to compete for the position of the best supplier.

Defense competition can take many forms. In fact, there may be no competition. For example, a **sole source** procurement might be selected due to the nature of the product and the availability of the source. A competition can involve two or more companies and may occur during research and development or implementation. Two generic forms of competition are used in military acquisitions:

- **Design competition**. An example of this was provided in the F-22 procurement discussion. Two competing teams of companies developed concept and design approaches, one of which was selected for the production contract. The benefits of this type of competition are: a clearer understanding of requirements through multiple perspectives; high risk items are identified and resolved in a more thorough manner; budget commitment is deferred until PDR; and the risk of selecting a poorly qualified FSD contractor (organization) is reduced.
- **Production competition**. Two or more companies bid to secure all or part of a production contract. Where more than one company is employed through initial production, risk is further reduced.

In general, you will be discouraged from dual awards for design and/ or partial production because this requires additional funding upfront. However, experience has shown that the longer two competing contractors are carried, the greater the opportunity for success. By avoiding program failure, and reducing the risk of schedule slippage and cost increases, overall program costs are generally less due to the continuing competition.

Concurrency/Time Phasing

Concurrency is a fast track acquisition strategy *[also discussed in Chapter 3, System Life Cycle and Methodologies]* that involves the overlapping of design, testing, production, and deployment activities. The overlapping and elimination of phases in the acquisition cycle, as well as overlapping or eliminating activities within a phase, are also choices based on the urgency of the need or the maturity of the system. A realistic **technology assessment** and allowance for **critical time duration** activities are key in planning a program with a high degree of concurrency between (or within) phases of the acquisition process.

Concurrency is used in response to a need to get a product to the field within a critical time frame. Short acquisition cycles, abbreviated proposals, condensed statements of work, minimal data reporting, use of commercial practices, and fewer reviews are all used to reduce costs and expedite schedules. A classic example where a concurrency strategy was successfully used was on the **Thor** missile program. The winning contractor's proposal consisted of a mere 20 pages describing how they intended to manage the program. The contract was awarded in December 1955 and the Thor flew successfully 13 months later.

Another example of a successful concurrency strategy was the **Single-Stage Rocket Technology (SSRT)** program. The contractor used commercial practices wherever possible which included almost one million lines of **COTS** test software for controlling ground and flight operations. Nearly 70,000 lines of onboard Ada flight control software were generated using a commercially available autocoding technique that cut costs an order of magnitude over conventional coding methods. They also reduced the number of government/ contractor program meetings which saved considerable costs and manpower. [WORDEN94]

An example of the risk involved in concurrency was the **Sergeant York** anti-aircraft gun or "DIVAD" for Division Air Defense, one of the most important weapons systems to be canceled while in production

in 1989. Concurrency was used to cut normal acquisition time (10+ years) in half and save money. This approach featured: parallel development by two competing contractors; the use of off-the-shelf components; a *skunk works* approach with thinly staffed government/ contractor program offices shielded from outside scrutiny; contractor flexibility in making cost/performance tradeoffs; limited and combined developmental and operational testing; and a concurrent follow-on development and initial production phase. The strategy stressed minimum government oversight during system development and reduced reviews and reporting requirements. Government access to contractor facilities and information was also limited. [GAO86] *[NOTE: Taken from Al Capp's Li'l Abner comic strip, the term "skunk works" was first used by Lockheed on the U-2 and SR-71 programs. It denotes a separate management operation outside the normal acquisition process due to the highly classified nature of the contractor's work.]* By concurrently developing, testing, and making improvements while in production the gun's considerable software integration problems never were ironed out. The procurement was canceled when it failed to perform during follow-on operational test and evaluation (FOT&E) and the Government was stuck 64 SGT Yorks at ~$42 million apiece.

The main advantage of concurrency is the achievement of an early operational capability. Another is that design maturity and operational problems surface sooner through earlier testing. Concurrency, however, introduces the substantial risk of performance shortfalls, schedule overruns, and cost growth, especially in complex, unprecedented software-intensive systems.

Design-to-Cost

The main advantage of **design-to-cost** is that it is a proven acquisition tool for obtaining lower unit costs. Design-to-cost forces identification of measurable design parameters which can be prioritized and used as targets in managing cost. This necessarily results in better requirements definition and increased efficiencies since budget and schedule limits are known upfront. The disadvantages are that it forces you to commit to a design-to-cost goal before final software requirements are defined. Hence, the need to sell the program may drive design-to-cost goals down to unrealistic levels. Also, since there are no practical ways to validate life cycle cost estimates, the contractor (or the Government) may choose to ***down-scope performance requirements*** to meet cost goals.

Computer Sciences Corporation (CSC) successfully used a design-to-cost approach to build a document control and tracking system for a pharmaceutical drug application. This was a process improvement initiative to invest in the development of a tool that would reduce the time it takes to get drugs through the testing and regulatory process (8 years) by 50%. This development had to be accomplished within a fixed time, fixed cost, and above a fixed functional baseline.

To accomplish their goal, they used what they called the "*Blue Chevy Policy*." All they needed was basic transportation — not a Mercedes, not a convertible, or a truck. They needed to provide an adequate solution today — not the ultimate solution tomorrow. Their design-to-cost approach relied heavily on hardware and software **COTS** products with as little custom development as possible. This required a partnership between the developer and the user to accomplish their common goal within mutually agreed upon architectural constraints. This meant the developers had to have direct and continuous access to the user, especially during the proof-of-concept prototyping process.

Training and documentation were developed in parallel with the system. When the program was completed on time, training began the next day. With a good architecture and user-involved prototyping throughout all phases of development, system functions exceeded expectations. They delivered a "*Loaded Blue Chevy*" and were able to grow the company into new areas of technology. [KEMP94]

Performance Demonstrations

An example of what can go right, is the strategy used by **F-22** planners to assess of performance requirements fulfillment. Using an open market strategy, each ATF competition team built their own demonstration aircraft using their own funds (50% contractor/50% government) and ideas which the Air Force evaluated during the Dem/Val phase. A less desirable alternative would have been a *paper design* — only testable at the end of full-scale development (FSD). [MRAZ91] This strategy was markedly different from that of the **B-2 Bomber** where the design was frozen at the beginning of Dem/Val. In effect, the demonstration phase, where alternative approaches are explored and bugs are worked out, was virtually skipped. This meant the B-2 was developed on an essentially noncompetitive basis with a cost-plus contract awarded to a single prime contractor eliminating the possibility of achieving a better design solution and reducing cost and schedule.

One reason ATF Dem/Val testing was such a success was because flight test objectives differed radically from traditional military testing. Instead of checking for compliance with a laundry list of requirements, ATF demonstration aircraft were used to show that the competitors had analytically predicted aircraft behavior by spot-checking the performance and technology issues each team thought were critical. "*We didn't come into the Dem/Val with what the military thinks of as requirements*," remarked **Lt. Col. W. Jay Jabour**, director of the ATF combined test force. "*We said to the contractors: 'Demonstrate what you think shows that you reduced risk to enter full-scale development.' Giving the contractor the latitude to fly his test program is a little bit different.*" [JABOUR91] **Colonel John M. (Mike) Borky**, former director of ATF avionics, noted that the Dem/Val phase was a huge success because it fostered rapid technology insertion and established the baseline configuration for the next phase, FSD. This method of testing along with demonstrations insured that once the selected design went into FSD, there would be no last minute surprises dragging out the schedule. There would also not be the need for additional large sums of money to fix a system that did not work. [BORKY91]

Performance Incentives

Performance incentives are a proven risk reducing acquisition strategy that rewards developers for exceptional contract performance. For incentive or award type contracts [i.e., **cost-plus award fee (CPAF)**, **cost-plus incentive fee (CPIF)**, and **firm-fixed-price incentive fee** [FFPIF], you may have difficulty identifying the factors on which to base the additional fee. While there is no standard guidance, incentive awards can be based on:

- **Milestone completion**. The degree of completeness for major software milestones has been successfully coupled with award fees. Surveys among Air Force managers show these award fees can enhance software development and documentation quality.

NOTE: Difficulties can arise if software development progress is evaluated independent of system development progress.

- **Software quality and reliability**. Quality and reliability are prime candidates for award or incentive fees since they greatly affect both development and support. Remember, software quality

and reliability can only be determined through an effective measurement program *[discussed in Chapter 8, Measurement and Metrics.]* [HUMPHREY90]

Contractors have a number of corporate goals which include profit, perpetuation, growth, and prestige. Most defense contractors are adverse to risk and operate on the premise that a satisfactory profit at acceptable risk is better than maximum profit at a high risk. Software development for the military is a very high risk corporate endeavor. **Performance incentives** and **award fee contracts** offer a means for motivating contractors to achieve more than minimal program objectives without excessive risk. This forces the Government and the contractor to work as a team — rather than as adversaries.

A CAUTION about this type approach is that often so much effort is put into preparing for the award fee board that productivity is sacrificed. Also, problems that should be dealt with as a team can be hidden from the Government to look good on performance reports. Nevertheless, award fees are a proven, effective means for assuring the achievement of desired performance, quality, and supportability objectives.

Make-or-Buy

The **contractor's make-or-buy decision** recognizes that few, if any, prime contractors can or want to make all of the many components required for a sophisticated, complex major software-intensive system in the time allowed, within cost limits, and at required quality levels. Buy decisions on the part of the prime can involve buying COTS or employing **subcontractors** to make subsystem software components. However, as you will learn in Chapter 13, *Contracting for Success*, subcontractors present government managers with a new set of issues. With subcontractors, the program office is divorced from direct contact management of the software developer because the subcontractor is under contract to the prime, not the Government.

The **government make-or-buy decision** involves whether to buy the software as COTS or contract for the development of a custom (or a combination of custom and COTS) solution. If custom-made, data rights issues must be analyzed and resolved.

Pre-planned Product Improvement (P3I)

If a technology or threat change occurs during the development of a software-intensive system, you can respond to these changes in one of two ways: (1) redesign the system to incorporate the change, or (2) continue the development as originally designed to deployment and modify the system later in the field. Both of these approaches can be costly to implement. There is always the risk that complete success in meeting unprecedented and unplanned for threats (or needs) will not be achieved. **P3I** provides an approach to meeting such needs without having to develop a new system. It entails making plans for probable future needs by improving the system as technology becomes available. The advantages of this strategy are:

- Responsiveness to threat changes and future technology development,
- Earlier initial operational capability (IOC) for the baselined system,
- Reduced development risk,
- Potential for subsystem competition,
- Enhanced operational capability for the final system, and
- Increased effective operational life.

The disadvantages are:

- Increased non-recurring cost during initial development,
- Increased technical requirements in areas, such as memory efficiency, source code efficiency, and reliability,
- Increased complexity in configuration management,
- Vulnerability to *goldplating* accusations and funding cuts,
- Compounding system management problems due to parallel developments, and
- Interference with the orderly development and implementation of effective support plans and procedures.

PROGRAM PLANNING PROCESS

In 1775, General Pierre de Bourcet profoundly influenced the art of strategic planning within the French Army and explained why planning is so important.

> *In war nothing is achieved except by calculation. Everything that is not soundly planned in its details yields no result.* [BOURCET75]

Planning is the fabric upon which a software program is woven, as planning underlies the whole software engineering process throughout the system life cycle. Planning includes the establishment of objectives and scope, consideration of alternative solutions, determination of required resources, and identification of technical and management constraints. Software development planning has historically been a poorly executed guessing game. The goal of planning should be to institute a management process that integrates the software engineering process with the estimating process.

The planning process involves **decomposition** of the system into functional elements or subsystems. The functional subsystems are then decomposed or allocated into lower tier elements. This process continues until the smallest functional element is identified. Within this systems engineering process, various **market analyses** and trade studies are conducted to determine the best solution to satisfy the particular subsystem allocated requirements or element of the subsystem. Normally, this process follows a hierarchy where the **architecture** is the first element to be evolved. The architecture is usually composed of components, interface between components and the functions to be interchanged. The architecture design also considers timing and bandwidth of the interfaces. From this architecture, the functional specifications for the components can be developed. These components will typically consist of both hardware and software.

NOTE: Open systems architecture is discussed in Chapter 2, *DoD Software Acquisition Environment.* Architectural design is discussed in Chapter 14, *Managing Software Development.*

The elements of a **strategic software management plan** are:

- **Objectives and scope**. The objectives identify the overall goals of the program, without consideration for how they are accomplished. The scope identifies the primary functions the software must accomplish, defined quantitatively. It describes what has to be done, for whom, by when, as well as the criteria for determining program success.
- **Risk assessment/management** *[discussed in detail in Chapter 6, Risk Management].* This activity filters throughout the process by determining in advance the possibility that a problem will occur, estimating its probability, evaluating its impact, and preparing solutions in advance. Risk assessment begins prior to acquisition strategy development and continues as an integral part of software management activities
- **Decomposition of software components by function and task**. Rather than attempting to understand and plan the entire program as a single entity, experience shows that it is easier and more effective to breakdown the overall program into smaller, more manageable elements. At a top level, the **System Segment Specification (SSS)** identifies those requirements and system functions that must be fulfilled by either software or hardware. At increasingly lower levels, the **work breakdown structure (WBS)** subdivides the program into more easily defined, understood, tracked and managed discrete tasks.
- **Market analysis**. This consists of trade studies for hardware and COTS software products. It also entails assessing competitive sources through **Sources Sought Announcements, Broad Agency Announcements (BAAs)**, and **Requests for Information (RFIs)** in the ***Commerce Business Daily***.

The following elements of the plan are discussed in the chapters listed.

- **Resource estimation** *[discussed in Chapter 8, Measurement and Metrics].* These are quantitative assessments on the number of people required and the cost, schedule, and size of each individual element comprising the whole software development task.
- **Software size estimates** *[discussed in Chapter 8, Measurement and Metrics].* These estimates are quantitative assessments of the amount of code required for each product element (system, subsystem, component, and/or module).

- **Software cost/schedule estimates** *[discussed in Chapter 8, Measurement and Metrics].* These estimates are based on the size of the program, product attributes (such as application complexity and security requirements) and environmental considerations (such as development team productivity, CASE tool use, and availability). Various estimating techniques and/or models can be used to estimate manpower requirements (staff-months of effort), cost ($), and schedule (calendar month duration).
- **Software support estimates** *[discussed in Chapter 8, Measurement and Metrics].* These are estimates of the resources required after system deployment. They are typically based on system size and original development effort. Although **actual** post-deployment costs must include product upgrades, error corrections, future evolutionary enhancements, and rehosting to new hardware platforms, most software support estimating models ***do not*** include all these elements in their estimates.
- **Progress measurement and control** *[discussed in Chapter 15, Managing Process Improvement].* These are on-going measures after contract award. They consist of formal programs for measuring and evaluating your contractors' progress against baselined budget, schedule, and quality standards. Typically, this involves defining and collecting specific metrics that are consistent with the agreed upon baseline. Although these measures may not answer all the questions about why variations from the baseline are being experienced, they should be sufficient to identify that significant deviations are occurring. They should also provide you with sufficient information with which to question your developer. Historically, ***this has been one of the weakest software management activities.***

ATTENTION! **The Strategic Software Management Plan need not be developed by the government program office except in outline form. However, one of the required products for evaluation during source selection should be detailed strategic software management plans submitted by the offerors with their proposals. Evaluate the quality of these strategic plans using information in the chapters identified in the above references.**

Planning Objectives

A veteran of the Vietnam War, Colonel Harry G. Summers, Jr. (retired) greatly influenced the serious study of strategic planning within the US Army. His theory on objectives was:

> *The first principle of war is the principle of* ***The Objective****. It is the first principle because all else flows from it. It is the strategic equivalent of the mission statement in tactics and we must subject it to the same rigorous analysis as we do the tactical mission.* [SUMMERS81]

The **objective** of the software planning process is much like the first principle of war. It is the first activity of the planning effort because all else flows from it. It provides a framework (or plan) from which an understanding of the mission and the execution of the effort can flow. This includes having a baseline upon which to estimate the resources, cost, and schedule required, as well as to evaluate recommendations for program changes. The plan serves as a guide for the SPO and as a means to communicate the content and execution of the program to individuals outside the SPO. As the program progresses, the objective and the plan must be subjected to rigorous analysis. You will progressively be able to quantify your original estimates. Thus, it is important to ***update your plan*** to reflect any changes in requirements and program management issues, and to more accurately document your new understanding of costs/schedules, risks, and other technical issues.

Planning Scope

The first activity in the planning process is to define the **scope** of the software effort. It includes function, performance, constraints, interfaces, and reliability. The definition process starts with the **System/Segment Specification (SSS)**, which is further decomposed with the **work breakdown structure (WBS)**. Make sure the functions described in the **Statement of Scope** are evaluated and refined to provide the greatest amount of detail before beginning your estimation process. This means that the performance allocated to the software segment during systems engineering must be bounded and stated explicitly (e.g., quantitative data such as the number of simultaneous users and the maximum allowable response time). **Constraints** and/or limitations (e.g., cost/weight factors that restrict memory size) are the limits placed on the software by external

hardware, available memory, or other existing systems. Mitigating factors (e.g., desired algorithms, well understood and available in Ada) must also be taken into account. [PRESSMAN92]

Recommendations for Program Planners

- Planners must allow for an extended schedule to compensate for the impacts of implementing new methodologies and adequately train personnel to build a repeatable process.
- Ensure there is enough schedule time to include quality in your product. Allow enough lead time between reviews and formal delivery of documentation.
- When new equipment, methods, and processes are introduced concurrently, a tremendous learning curve exists. Bringing personnel into a program after it starts makes it difficult for them to grasp the scope of the program and become fully productive. Therefore, bring on key personnel as early as possible in accordance with the staffing plan. Any other new personnel should be phased in incrementally so as not to interfere with total team productivity and cohesion.
- Scheduling for an integrated government/contractor team must involve both government and contractor management.
- On a program involving new processes, equipment, and methodologies, development personnel must be allowed time to come up to speed on goals, tasks, and associated start and completion dates. This can be accomplished by providing each team member with inchstone schedules that are planned and staffed 60 days in advance. These should include all known leaves, vacations, training, holidays, commander's calls, etc.
- To avoid a series of nontrivial changes during software analysis and design, iron out Interface Requirement Agreements (IRAs) as early as possible. Interfaces can be identified as a technical risk. Organize a Risk Management Working Group to oversee IRAs and a risk mitigation officer to monitor and plan risk mitigation.
- Make sure task configuration management (CM) personnel are briefed on CM policies, procedures, and the possible consequences of not following them. CM policies and procedures should be included in the SDP and distributed as required reading to all task members.
- The configuration requirements, implementation, and maintenance for a local area network (LAN) requires planning and dedicated, specialized personnel. Therefore, upfront analysis and planning to determine LAN requirements is necessary with special consideration

paid to how much and what type terminal equipment will be connected to the LAN. Include communications specialists in your staffing plan.

- Adequate identification of hardware and software constraints, connectivity, and supporting documentation must be planned for prior to starting a task. Proper planning of staffing and tools for each task lessens the impact on productivity.
- Government staffing must be coordinated between the task leader and his/her government counterpart to ensure appropriate skills are provided at proper times in the life cycle. Have each task leader identify a staffing strategy. The technical quality assurance evaluator (TQAE) should then review this strategy and plan a similar strategy in coordination with the task leader.
- An external interface process should be documented in a separate appendix to the SDP.
- Although not usually scheduled for delivery, prototype development requires a good support environment. This includes the hardware environment and regular system backup. These elements, often taken for granted, should be routinely provided in development deliverables.
- Documentation must be available from the onset of any new engineering endeavor. Prototyping language manuals and texts will pay for themselves almost immediately after acquisition, as they eliminate the time-consuming trial and error approach to learning.

PROGRAM DECOMPOSITION

In Chapter 4, *Engineering Software-Intensive Systems*, you learned that one reason for employing the principles of software engineering is that it provides a method for handling **complexity**. This is accomplished by applying the old adage of *divide-and-conquer*. Major software developments must be decomposed (broken down into manageable parts) to enable realistic estimates of size, time, and manpower. Methods of decomposition differ depending on program objectives, which may be based on either function or design. **Functional decomposition** divides the program into basic components from a user's perspective; whereas, **design decomposition** divides it into software components or modules. [BENNATAN92] Your first layer of decomposition is at the system level with the SSS. From that the WBS is developed.

System/Segment Specification (SSS)

The most important and critical aspect of system development is to nail down function-level system requirements before they are allocated to either hardware or software components. For weapons systems, the SSS containing general software requirements, is an initial method of decomposition at the system level. In the SSS, system requirements are defined in quantifiable, measurable, and testable terms. [DSMC90] The SSS is then the basis for further decomposition into the WBS.

Work Breakdown Structure (WBS)

A WBS should be developed for each major acquisition program (or major modifications thereof), and for each individual contract within the program. The WBS, in its various forms, can serve as a useful tool for planning, control, and communication throughout your program. The WBS, if sufficiently written, defines the program's total objectives and relates the many work efforts to the overall system. The WBS is the foundation for:

- Program and technical planning,
- Technical description of program pieces,
- Cost estimation and budget formulation,
- Schedule definition,
- Statements of Work and specification of contract line items,
- Progress status reporting and problem analysis,
- Tracking of technical changes [Engineering Change Proposals (ECPs)], and
- Engineering management.

WBS Interrelationships

The **summary WBS** defines the upper three levels of a system. The **project summary WBS** is tailored to a specific program or project. The **contract WBS** defines the complete work effort for a particular contract or other procurement action. It contains applicable portions of the project summary WBS plus the extension of any levels necessary for planning and control. The interrelationship between WBSs is illustrated in Figure 12-2 (below).

From the project summary WBS, individual contract (or development organization) WBSs can then be developed. The program office can initiate preliminary contract WBSs before contract award that contain contract line items, configuration items, contract specifications, and

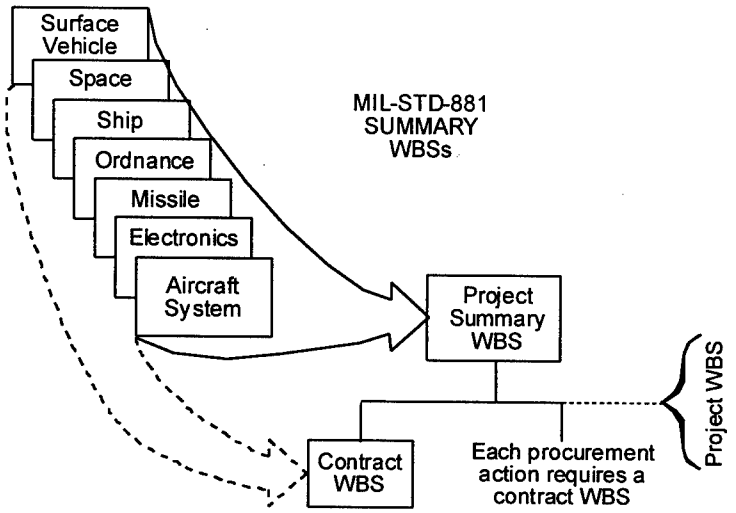


Figure 12-2 Interrelationships Among WBS Types

industry responses to the draft RFP. The initial project summary WBS and first contract WBS must be established at the award of the first development contract. As the program progresses and additional contracts are let, the project WBS must be extended to all levels it addresses, but the basic structure should remain unchanged. A single project WBS, with element nomenclature and definitions, should be maintained throughout the acquisition process to ensure traceability. The components of the WBS are:

- **Prime mission product**. The prime mission product element is the hardware and software used to accomplish the primary mission of a defense materiel item. It includes all integration, assembly, test, and checkout, as well as, all the technical and management activities associated with individual hardware and software items.
- **WBS element**. This describes a discrete portion of a WBS that is either an identifiable item of hardware, a set of data, or a service. An element can consist of one or many work packages.
- **Subsystem**. This refers to all the hardware and software components of a subsystem.
- **Software component**. This is all the software integral to any specific subsystem specification and can be an aggregate of application and system software *[discussed below]*. (It excludes software specifically designed and developed for system test and evaluation.)
- **Work package**. This represents the work to be performed at the lowest WBS level where work performance is managed. Developed

by the contractor, it defines the work, how its accomplishment is measured, how it is tied to a schedule, and where responsibility lies for production of the operating unit. Interrelating the who, what, when, and how much for any task effort, it is the heart of management control and provides visibility at designated levels. Program performance can be measured and controlled by monitoring reports on the technical and schedule aspects of each work package or combination of work packages.

Prime Mission Product Summary WBS

The **prime mission product summary WBS** identifies the upper three levels of a WBS and defines the top-level software elements and their placement in the structure. The prime mission product summary WBS is used to develop the software project summary WBS.

Software Project Summary WBS

A **software project summary WBS** is usually the result of the systems engineering efforts conducted during Concept Exploration. At this time, the most suitable summary WBS software items are considered that best satisfy operational needs. The preliminary software summary WBS should not be constraining, but evolves and is tailored as program objectives stabilize. Developers should be encouraged to propose changes to the preliminary project summary through creative, alternative development options. The software project summary WBS elements are:

- **Software WBS elements**. Software WBS elements are described generically and apply to each type of defense system. The associated activities and deliverables for which cost data are collected are listed with each software WBS description.
- **Application software**. Application software is specifically developed for the functional use of a computer system. Examples are battle management, weapons control, and data base management software. This element refers to all the effort required to design, develop, integrate, and checkout prime mission product applications, builds, and CSCIs. It excludes all software integral to any specific hardware subsystem specification. Figure 12-3 (below) illustrates the breakdown of both application and system software CSCIs.
- **System software**. System (or support) software is designed for a specific software system, or family of software systems, to facilitate its (and its associated applications; i.e., operating systems, compilers, and utilities) development, operation, and maintenance.

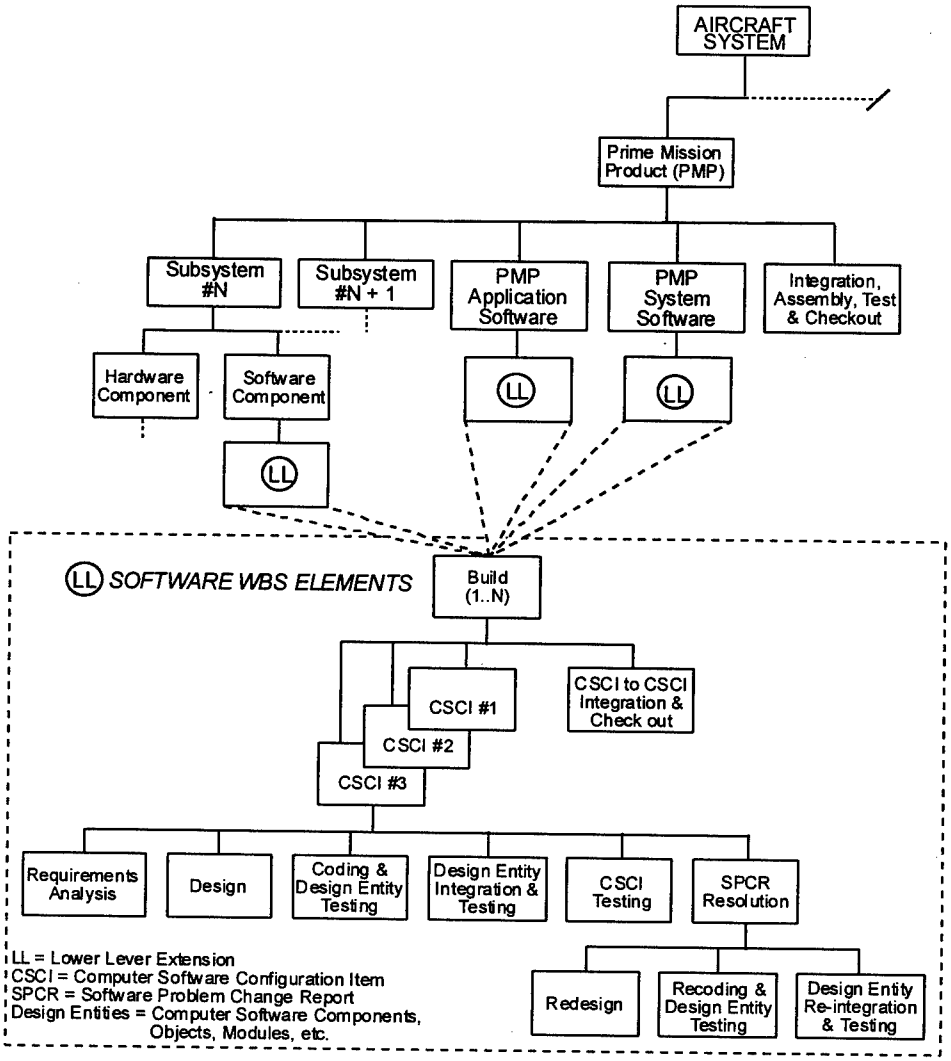


Figure 12-3 Interrelationships Among WBS Types

It also includes all the effort required to design, develop, integrate, and checkout the system software, including all software developed to support any prime mission product software development. It can also include multiple builds.

Software Contract WBS

Only one preliminary **software contract WBS** is used for each RFP and its ensuing contract WBS. The program office structures a preliminary contract WBS by selecting elements of the approved project summary WBS that apply to that contract. It then organizes them into a framework supporting the approved project summary WBS and development objectives. Software subsystems may then be extended to the next lower level. Traceable summarization of individual contract WBS(s) into the approved project summary WBS are maintained. The contract WBS does not need to completely mirror the project WBS. For contracting issues (e.g., cost accounting) a WBS different from the project tracking WBS may be necessary. The functional integration of the project summary WBS with the contract WBS is illustrated in Figure 12-4 (below).

In their proposals, or during source selection, offerors are encouraged suggest changes to certain elements to make the contract WBS more effective. These changes are approved by the government program manager. The final contract WBS, based on the contractor's proposal, suggested changes, and contract negotiations, becomes the basis for a more detailed definition necessary to manage the effort. The contractor must also prepare program-specific terminology and definitions for extended elements of the contract WBS.

A couple of points must be emphasized. First, the contract WBS provides the link between the contracted effort and the overall program to include description of the interfaces necessary to integrate the software of one contractor with that of other contractors or agencies. This is to ensure that all software being developed is compatible when integrated with other software and hardware at the next higher level of integration. Second, be careful to select WBS elements that permit structuring of budgets and tracking of costs to whatever level is necessary for control.

You can accomplish this by assigning **job orders** (or customer orders) to the cost account level for in-house efforts and by structuring line items (CDRLs) or work assignments *[discussed in Chapter 13, Contracting for Success]* in accordance with the WBS. Usually, a cost account is established at the lowest level of the contract WBS where costs are recorded and compared with budgeted costs. This cost account (WBS element) is a natural control point for cost/ schedule planning as it is the responsibility of a single organizational element. Contractors should maintain records to the work package

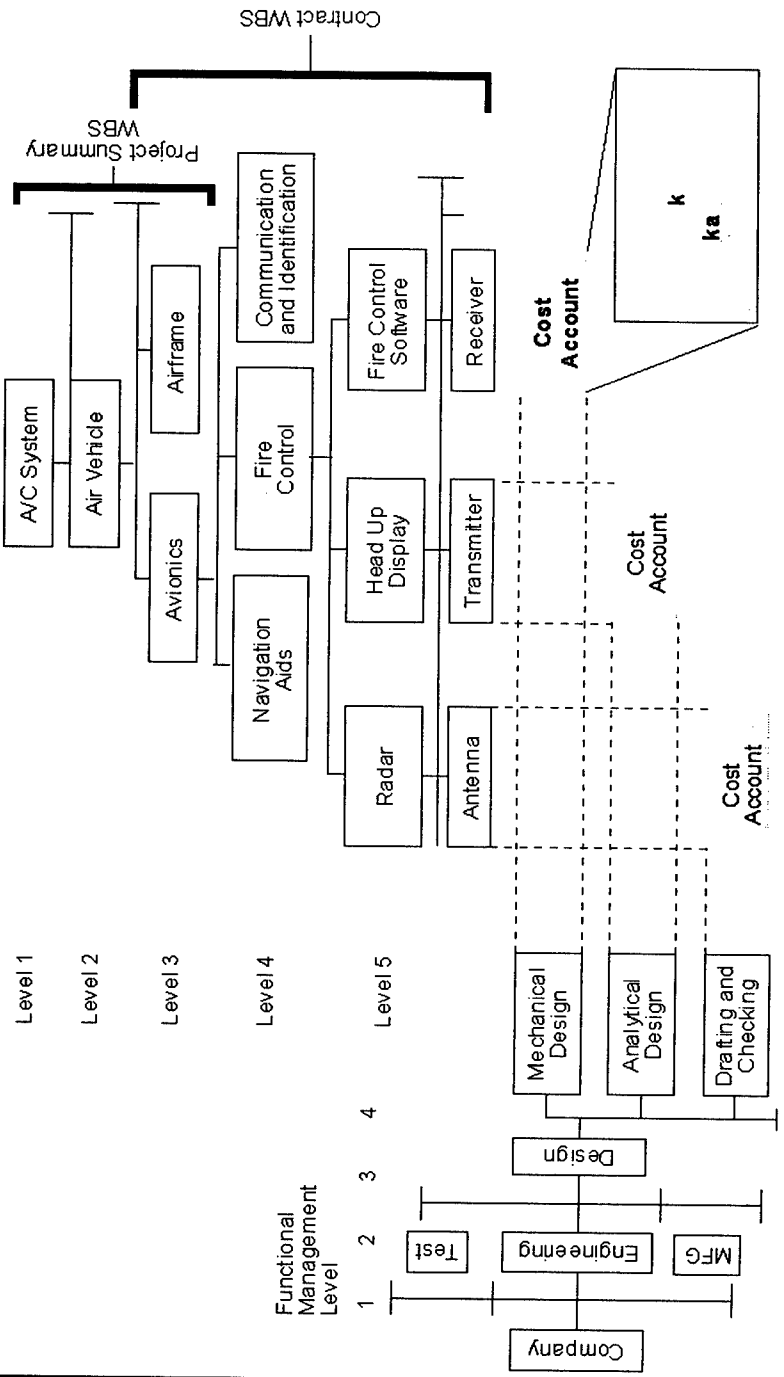


Figure 12-4 Software Project and Contract WBS Functional Integration

level so the Government has visibility to the cost account level. ***Ideally, you and the contractor will agree on a WBS which is integral to [and not disruptive of] their development process that they would normally use for internal tracking and management.***

Software Project WBS

The software program/project office prepares the **software project WBS** by compiling the elements of the extended contract WBS(s) with the project summary WBS. The program office then incorporates the levels of the extended contract WBS(s) it considers necessary for program management and other related requirements into the project WBS. This compilation occurs as successive extensions of the individual contract WBS(s) are identified throughout the program. The formal project WBS is completed prior to initiation of the system integration and test phase. A 3-level project summary WBS for the **F-22** is illustrated in Figure 12-5.

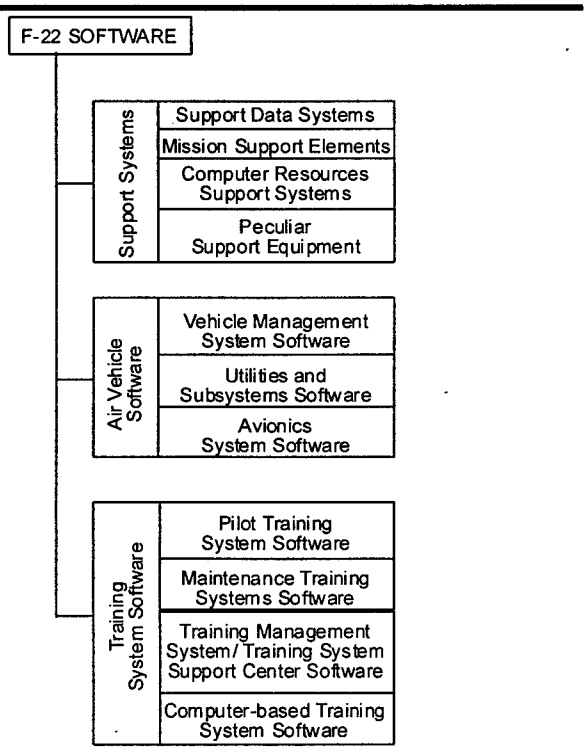


Figure 12-5 F-22 3-level WBS

The **Integrated Master Plan/Integrated Master Schedule** concept that evolved on the ATF program is one approach to force contractors to perform detailed, step-by-step planning and report progress and costs against their plan. If contractors are held accountable to deliver functions rather than configuration items, they cannot cheat on software performance and integration because delivering the hardware alone will not trigger payment. This is something of an oversimplification, but the basic concept of requiring that contractors deliver **integrated system capabilities** and minimizing progress payments for arrival on dock of a system capability, rather than bits and pieces, is highly recommended.

WARNING TO WEAPON SYSTEM PROGRAMS! The lack of a software WBS has been the Achilles Heel of many weapon programs. Do not be caught without meaningful insight into your highest risk area.

MARKET ANALYSIS

The **Defense Standards Improvement Council (DSIC)** states that **market analysis** is key to meeting the spirit and letter of **Secretary Perry's** June 1994 Memo. *[See Chapter 2, DoD Software Acquisition Environment, and Volume 2, Appendix C.]* You should perform a market analysis to determine if commercial products are available that meet your identified need because current MilSpec reforms make aggressive market analysis imperative.

Through comprehensive market analysis, you will be able to ascertain if adequate commercial product alternatives exist and to identify satisfactory replacements for software MilSpecs and MilStds. If your market analysis illustrates that certain software MilSpecs or -Stds can meet your identified need, your analysis results can serve as the basis for a waiver request to the **Milestone Decision Authority (MDA),** or you can cite the MilSpecs or MilStds as guides without mandating that they be literally followed.

Whether your program is a new-start, an on-going, or one in PDSS, you must perform a market analysis prior to every requirements definition effort. The data you collect during the market survey are then used to reassess your original requirement. You must determine whether a modification to the original requirement will result in greater overall value to the Government in terms of cost, performance,

availability, reliability — or other risk drivers you have identified. You market survey should also cover maintenance and support data, test results, and user satisfaction analyses. These data are used in developing your support strategy and the TEMP. *[The SD-5, "Market Analysis for Nondevelopmental Items," Assistant Secretary of Defense (Economic Security) [OASD/(ES)] describes a generic approach for market analysis. A training program is also available from OASD/(ES). See Volume 2, Appendices A and B for a point of contact.]*

Software Product Definition and Decomposition

Software product definition and decomposition will be complete once you have accomplished the basic planning process discussed above. Your product will be identified and decomposed, at least initially, through the SSS and the various WBSs. However, it may be necessary to modify or adapt these items to your software cost, schedule, resources, and support estimate preparation requirements.

BASELINE ESTIMATES

The basic software estimating process mirrors the strategic planning process and builds on and supports many of the other planning steps. It consists in defining what will be estimated, breaking the total effort into appropriate lower-level elements, determining the scope (size) of each element, assessing the software development environment, and performing assessments of alternatives and risk factors. Once these elements have been quantified, evaluated, and boundaries placed around their values, **baseline estimates** of cost, schedule, resources, and support can be determined and assessed. Table 12-1 (below) from Kile's *A Process View of Software Estimation* outlines the steps necessary for bid preparation. Although it names the steps and presents the view differently than discussed here, the basic process is the same. [KILE91]

The software estimating process is an interactive, dynamic process. As program requirements, the development environment, and/or the program funding profile change, ***re-estimation of the effort/cost and schedule must be performed.*** Contractor ECPs must also be evaluated for their affect on both development and support costs, as well as schedule. Funding constraints typically result in program delays which can, in turn, increase cost.

PHASE	MAJOR ACTIVITY	SPECIFIC PRODUCTS
1. Design Baseline	Define a point of sufficient precision to identify the number of CSCIs and the required functionality of each.	List of CSCIs, functionality, and similar completed projects or CSCIs.
2. Size Baseline	Using the products from the Design phase, define the expected size for each CSCI.	List of CSCIs with appropriate size information.
3. Environmental Baseline	Using the products from the two previous phases, determine the environmental characteristics required and available to perform the job.	List of software cost model parameters and their initial settings along with a written rationale for each.
4. Software Baseline Estimate	Using the size and environment products, make a software cost model run (using whatever model best satisfies the organization's needs).	Output from the software cost model showing schedule and cost information.
5. Project Baseline	Using the output from the Software Baseline Estimate phase, add those elements not included in the particular software cost model (each model has a specific set of items not included in the estimate) and subtract those elements excluded from this project.	A complete estimate of the costs and schedule for the software portion of the project.
6. Risk Analysis	Determine the cost/schedule risk associated with the Project Estimate. Make changes to the size or environment products to perform what-if analyses. Determine the size and/or environment setting required to validate the final software bid.	Risk assessment, risk graphs, risk memorandum with Parameter-by-Parameter risk explanations.
7. Project Bid	Perform analysis of the Project Estimate, considering such factors as expected competition, type of contract, budgetary or personnel constraints, risk analysis, etc. Convert labor and ODC estimates into contractor's price and determine the Project Bid.	Project Bid.
8. Dynamic Cost Projection	Using existing known environment and size information, produce a revised Project Estimate and determine the remaining costs and schedule-to-complete for the on-going project.	Cost-to Complete, Schedule-to-Complete, Size-to-Cost.

Table 12-1 The Software Estimation Process

To develop master schedules, acquisition strategies, and preliminary budgets, a preliminary cost/schedule estimate reflecting the **program baseline** needs to be developed using preliminary size and environmental assessments *[discussed in Chapter 8, Measurement and Metrics]*. This baseline estimate provides a starting point from which alternatives may be compared and changes tracked. Throughout development, as assessments are updated to reflect current conditions, cost/schedule estimates must be updated to support decision making at all levels. A cost track from the baseline estimate to each update,

as well as clear, understandable documentation substantiates the need for programmatic change. *[A rule of thumb for a well-documented estimate is that it is verified by a second party.]* The *goodness* of an estimate depends on whether factor assessments are realistic, appropriate risk is considered, and estimating methodologies substantiate reasonableness of the cost estimate. Significant cost and schedule *drivers* should be re-estimated and documented using a secondary methodology as a confidence check. A minimal confidence (or sanity) check, is performed on significant cost elements to assure that the estimation is within an acceptable range of general knowledge (e.g., SLOC/staff month is within range of similar software programs).

NOTE: An overview of several estimating techniques/ methodologies is found in Chapter 8, *Measurement and Metrics*.

The preliminary estimate becomes the **baseline** from which the process of updating your estimates proceeds, and continues throughout the development life cycle. As program knowledge increases, metrics data are collected and analyzed, and your estimates are updated, your cost and schedule estimates will become increasingly more accurate. This approach does not omit or conflict with longer term acquisition strategies such as systematic reuse and families of product-lines and systems. Long term, wide scope acquisition planning is necessary to ensure a cost-effective acquisition, especially when such requirements may contradict the profit motives of individual development contractors.

Once an initial cost and schedule estimate has been developed, significant effort is required to analyze and understand the estimate before accepting it as a formal part of your strategic plan. The analysis of your original estimate serves four purposes: (1) to make sure the estimate is thoroughly understood, (2) to insure that the estimate is as accurate as possible; (3) to provide a baseline upon which to evaluate programmatic alternatives (e.g., trade studies, software capability assessments, tradeoff analyses, and development methodologies), and (4) to conduct risk analyses. The original estimate analysis includes answering the following:

- Does the estimate make sense?
- Are estimated schedules, costs, and effort consistent with prior experience?
- Does the estimated effort, cost, and schedule meet programmatic requirements?

- Are required productivity levels reasonable?
- Have all relevant costs been included?
- Have any cost elements accidentally been included more than once because different estimating techniques were used for different WBS elements?

Estimation Accuracy

The accuracy of your cost estimate directly relates to the quality of the information upon which it is based. The exactness of this data increases as a function of time and the stability of requirements. During the planning phase of a program, requirement uncertainties often result in questionable estimates. As time progresses, the fidelity of the information improves along with the accuracy of the estimates. [MARCINIAK90] The quality of the information is also dependent on the skill and experience of your analysts/engineers who gather and analyze the input information. The quality of the estimate is, similarly, dependent on their skills and experience in software cost and schedule estimating, the specific estimating methods and models used, as well as their familiarity with the software system being estimated.

Using a second (or possibly third) estimating technique or model to identify these potential problems is a proven, effective way to compare estimation results (after normalizing results for equivalent content). To correct any identified problems, your cost analyst must change model input settings *[not input data]* to reflect a better understanding of the information required by the model and/or to seek additional clarifying data upon which to base changes in model inputs. If your model's estimated schedule exceeds programmatic requirements, your analyst may need to *turn-on* a **schedule constraint variable** within the model. If the staffing profile predicted by the model is inconsistent with the development plan, a **staffing constraint variable** may need to be adjusted. After several iterations (under a variety of assumptions and with varying parameter settings) your analyst should arrive at an estimate that is both credible and reasonably accurate. This estimate should also include a risk assessment. At this point, the estimate may consist of a **range of estimates** that reflect different assumptions and probabilities of success rather than a **single-point-estimate**. You should review these estimates and risk assessments and provide additional guidance for further analysis and/or approval.

NOTE: Parametric cost/schedule models are discussed in Chapter 10, *Software Tools*.

Once the estimate is as accurate as possible using known information, it can be used to perform sensitivity analyses, risk analyses, and *"what-if"* exercises by varying model inputs based on expectations or alternative sets of assumptions. If there is uncertainty about the size measurement (or other factors influencing the cost or schedule), high and low end estimates of the expected range should be developed. These studies can be used to identify risk areas and to develop contingency plans. If there are constraints on your budget or schedule, your estimate should be derived taking these limitations into account. This will provide your **baseline estimate** as to the viability of developing the software within identified constraints. If it appears that either cost, schedule, or resource limitations can not be met, other programmatic options must be examined during planning, rather than waiting until these constraints have been violated. ***It is extremely important that you do not change estimate parameters to meet programmatic restrictions.*** This will merely invalidate your estimate, your program planning activities, and greatly reduce your likelihood for program success.

CAUTION! Beware of "*Rosy Scenario*" (also known as "*Optimism*"). Program managers are inclined to manipulate the input variables to software estimating models to assure an "*acceptable*" outcome in terms of estimated cost and schedule. One example is to understate the size of the program, while another typical situation is to overstate the capabilities and/or resources (tools and practices) of the development team. Time and again this has led to broken programs, delays, restarts, loss of confidence, all around embarrassment, and on occasion — program cancellations. Well-documented estimates using reasonable, but conservative, assumptions will bring you accolades in the long run, even if there are grimaces and groans in the near term about the predicted cost and schedules taking too long. The only acceptable means of reducing schedule are the use of appropriate CASE tools (e.g., code generators), where appropriate, increasing the reuse of software assets from repositories, and using incremental or evolutionary

development approaches which provide nearer term usable products while maintaining program credibility.

Program Estimate Selection

After the analysis of the estimate is complete, it is up to you to select the cost and schedule estimate baselines which become part of your **Strategic Plan**. Ideally, this is performed in cooperation with your development team manager as an on-going activity in your program management process. This baseline must be periodically updated to reflect changes in the Strategic Plan as more is known about your program.

The key element in selecting the baseline estimate is the level of cost and schedule risk you are willing to accept. You need to understand that, although it may be possible to accomplish your program within the cost and schedule to which you have committed, based on historical examples there may be only a 10% probability you will succeed. It is only through realistic estimates and early planning that this probability can be increased. Similarly, if the schedule for a software development is dictated by other mission-critical factors (such as a payload launch date), you must realize and understand the probability of meeting that schedule. Once you understand your probability of success, you can rethink your strategy by planning various incremental efforts to insure that critically-necessary functionality is completed on time. Other noncritical functionality may be deferred to later development stages, or if necessary — omitted completely. Software estimating models (used in concert with independent risk analysis techniques) should be used in assessing the critical cost and schedule risks associated with changes in your program.

CONTINUOUS PROGRAM PLANNING

The final and most challenging step in planning is for you to constantly re-implement the planning process throughout the life of your program. Budget cuts, personnel cuts, short schedules, incessantly changing requirements, and the development environment force you to continuously re-evaluate your estimates. To ensure successful program completion, you need to update your **Acquisition Plan**. Figure 12-6 illustrates how planning is a continuous, iterative process.

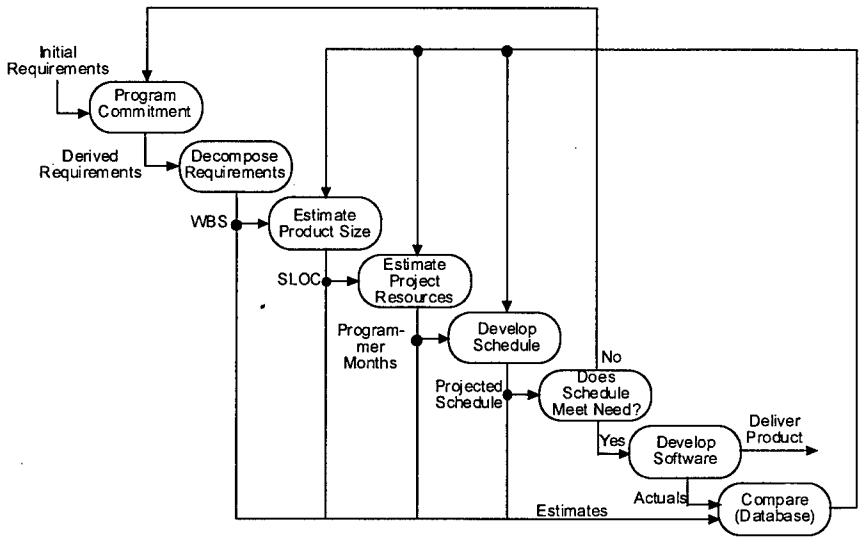


Figure 12-6 Iterative Software Planning Process [HUMPHREY89]

Remember, ***the requirement to prepare software cost and schedule projections does not end.*** The baseline estimate must be updated to reflect changes in your program environment, your increased program understanding, and the actual metric data being collected. Similarly, cost and schedule impacts of proposed and unforeseen changes can be quantitatively evaluated using estimating models and your baseline estimate. These may not answer all the questions you have about what is happening in your program, but they will provide you with a solid starting point from which you can direct your questions.

Software estimation (and indeed software development) is still an art practiced with varying levels of standard procedures, tools, and methods. Also, there are many unknown and dynamic variables (i.e., human, technical, and political) in the development process that affect the software effort. For example, requirements are frequently added or changed well into the software testing phase. Because requirements frequently change, there is an axiom in the cost analysis community that states: ***"original estimates are never correct because we never build what was originally estimated."*** Also, initial estimates of software size are typically based on limited information and are often driven by *optimistic, rosy scenario,* success-oriented influences. Software cost and schedule estimates also fall-short because the analyst performing the estimate is unfamiliar with either the estimating model(s) being used or the specific details

of the program — or both. As a result, estimating the cost, effort, and schedule of software development is a necessary but inexact science. Software program estimates can be improved, however, by using a systematic, disciplined estimation procedures.

Continuous Planning Recommendations

- No one likes to be a slave to schedule, but adherence to schedule promotes program stability and inhibits requirements creep. Develop your deliverables around the delivery of a functional capability and use cost payments as incentives to meet those deliveries.
- If a software development effort begins to sink because of schedule slips, throwing more people onboard may not help. In fact, more people can actually cause the program to fall even farther behind due to added communications and training requirements that decrease productivity. Therefore, plan team composition and buildup in the early strategic planning phases.
- Initially, **QUALITY** does not stand out like this. If you do not think quality upfront, you will pay dearly for its neglect later.
- If you are living with evolutionary requirements, it makes sense to pursue an evolutionary acquisition strategy. Develop an acquisition strategy that is flexible, can accommodate evolutionary change, and deal with risk.
- Software planning is an iterative and continuous process. Initial estimates must be refreshed and reflected in updated schedules and resource commitments.
- Monetary reward is a proven incentive for contractors to produce quality software. Money might not be everything, but it sure is way ahead of whatever is in second place.
- Software size estimates have traditionally been poor and rank right up there with estimates on the Gross National Product. Until recently, inaccurate estimates of SLOC have been a major impediment to accurate software development cost estimates. Another major impediment has been the failure to accurately estimate the capability of the development team.
- To make your estimates more accurate, use a combination of estimation techniques. Of all the program risks with which you must deal, size estimating will be your biggest planning problem. A word of caution is to use well-documented estimates based on conservative assumptions.
- Use the Mitre Skills Matrix *[discussed in Chapter 13, Contracting for Success]* to reduce the team capabilities risk element.
- Identify software support requirements, as well as computer hardware support requirements, in program budgets and schedules.

- A well-planned measurement program *[discussed in Chapter 8, Measurement and Metrics]* is an investment in successful management and product quality.
- Performance risk can be reduced by planning for delivery of incremental levels of functionality.
- Use program decomposition and program management automated tools to get a handle on program complexity.
- When comparing estimates produced by different cost models, make sure they have the same definitions of environmental input parameters.

OTHER PLANNING CONSIDERATIONS

There are several other areas of strategic planning that you must include in your planning process. These include: the use of milestones and baselines to track program progress towards achieving objectives, factoring in the often hidden (but often substantial) cost of software scrap and rework in your estimates, program budgeting and funding considerations, upfront definition of requirements for software safety and security, and planning for future changes in technology that can impact your development efforts.

Major Milestones and Baselines

Milestones signify major events in the software development process. The completion of requirements and design specifications are major milestone events. Major program milestones often gain added importance through their linkage to other events, such as budget payments used as measures of program progress or for determining baselines. If milestones can be described as major program events, then **baselines** can be described as major milestones. [BENNATAN92]

The IEEE definition of a **baseline** is "*a formally agreed upon specification that serves as the basis for further development.*" [IEEE87] Baselines are important in DoD software development as they indicate critical times when major milestones are finalized. Baselines also provide significant and complementary ways to control acquisition programs. **Strategic planning baselines** include:

- **Cost/schedule control performance measurement baseline**. This baseline provides the budgeted cost of work scheduled and is the measure against which schedule and cost variances are calculated.

- **Configuration management baselines**. The software configuration management process *[discussed in Chapter 9]* is important in providing support to the baselining of system products, and is central for controlling the development process. Baselines that mark the completion of major milestone activities are **formal baselines**. Changes to formal system baselines can directly impact both cost and schedule. With formal control, any changes to the baselined system must be approved by the authority responsible for system integrity as defined in that baseline. In software development, there are three formal baselines.
  - **Functional baseline**. The functional baseline establishes the requirements the system must satisfy. With functional baseline establishment, system specifications are placed under control.
  - **Allocated baseline**. The allocated baseline marks the end of the software analysis phase. The allocated baseline is established when requirements are allocated to individual software subsystems. It captures the linkage between the architecture and software requirements.
  - **Product baseline**. A product baseline is established when the software system is fully designed, developed, and tested. This baseline defines the produced software product, and provides the framework for modifying the system through defect correction and incorporation of new requirements.
- **Acquisition program baseline (APB)**. The APB provides quantifiable targets for key performance, cost, and schedule parameters of an acquisition program throughout the acquisition process phases. The APB has two components for each parameter, an **objective** and a **threshold**. Objectives and thresholds are determined differently for cost, schedule, and performance. The user's **Operational Readiness Document (ORD)** provides performance objectives and thresholds. The ORD also provides the user's requirement for **initial operational capability (IOC)** and **full operational capability (FOC)** — both of which have schedule implications. Cost and schedule objectives and thresholds are developed by your acquisition team. *[Recommendations for ORD preparation are found in Volume 2, Appendix N.]* APBs are sequentially refined as we move through the life cycle phases and are submitted at Milestones I, II, and III. The APB may be adjusted at milestone approval (or program reviews) based on changes in requirements and/or on the results of activities taking place in the previous phase. The APB can also be adjusted in response to a **baseline breach**. Only those performance, schedule, and cost parameters attributable to the breach, however, can be adjusted.

- **Operational performance thresholds** are the user's minimum acceptable requirement for the system when fielded and are derived directly from the ORD. (Other performance thresholds may be added by the **Milestone Decision Authority**.) Cost objectives and thresholds should reflect the **independent cost estimate (ICE)** for the program to meet performance objectives. For schedule, the objective is the most likely date for a key event (such as a milestone review, design review, or the completion of a test activity).

When the operating command identifies an unfulfilled need, they must start working closely with the developing command and the supporting command in defining system **thresholds** (*minimally acceptable requirements*) and **objectives**. This approach recognizes that technology, funding, or schedule may preclude the developing command (and its contractors) from achieving each and every objective. The objectives, if properly integrated into the program, can help the system designer accommodate **P3I**. Thereby, parallel development upgrades can be incorporated at appropriate procurement stages. A better understanding of requirements and more effective teamwork can be achieved while still maintaining acquisition competition.

Program executive officers and senior acquisition executives must establish an atmosphere that fosters frank program assessments by the development team. One method is to ***actively support realistic realignment of system requirements and schedules during program reviews***. This must be a strategic part of the iterative process of refining the system by considering technology, budget, and schedule. **Program baseline documents** (e.g., requirements correlation matrices and system maturity matrices) must mature throughout the evolutionary process and be used as management tools. They should reflect the refinement of system requirements as the development proceeds, provide an audit trail of requirements, and establish the rationale behind subsequent requirements changes.

During each phase of development, you need to maintain a **current estimate** of cost, schedule, and performance parameters. If the current estimate indicates a threshold breach is anticipated, or has occurred, they must be reported and acted upon. The APB complements the related concepts of the **maturity growth curve**, **exit criteria**, **event-based contracting**, and an **event-driven Acquisition Strategy**. Figure 12-7 (below) illustrates how these concepts are complementary. A performance parameter or capability (such as

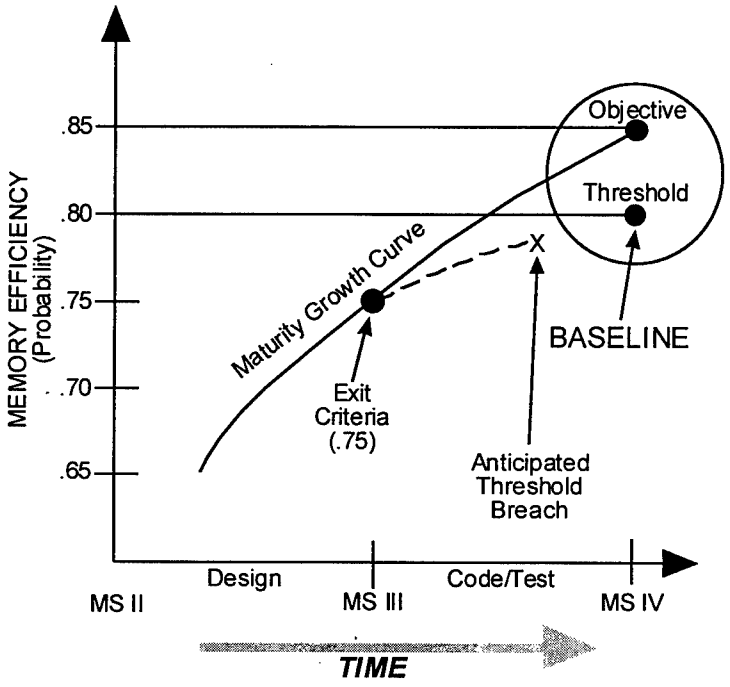


Figure 12-7 Maturity Growth Curve

code, memory, input/output efficiency, or reliability level) is the baselined parameter. *[**Memory efficiency** is depicted in this example.]* The baseline is represented by a threshold (minimum requirement) and the objective (desired level). In some cases the objective and threshold may be the same value. However, if deltas exist between the objective and the threshold, the deltas should provide room for tradeoffs in performance, cost, and schedule parameters.

Developmental testing (DT) during Phase I tests to values along the diagonal line, called the maturity growth curve. In Phase II, DT is focused on measuring the attainment of contract specification values, whereas operational testing is structured to demonstrate the threshold value (i.e., the minimum requirement). [DSMC92]

Program Budgeting and Funding

The Air Force develops its programs through the **Planning, Programming, and Budgeting System (PPBS)**. It consists of three parts:

- **Planning** identifies the threat facing the nation for the next 5-15 years, assesses our capability to counter it, and recommends forces necessary to defeat it.
- **Programming** allocates resources for competing requirements within the fiscal and manpower ceilings imposed by the Congress. This effort develops five-year program, i.e., the **Program Objective Memorandum (POM)**.
- **Budgeting** provides the initial estimated cost of approved plans and programs and refines estimated costs as programs are better defined or modified in subsequent POM cycles, **budget estimate submissions (BESs)**, or the **President's Budget (PB)**. *[Refer to AFP 172-4, The Air Force Budget Process.]*

As a program manager, your role in the PPBS process is important. You may not be involved in the initial planning process, but you are an important player in the software-intensive systems programmed as a result of this planning. You must investigate the technology and software solution recommended to satisfy the planning requirement, and if the solution is not sound, you must bring that to the attention of all concerned and work to resolve all issues. These efforts are critical to the success of your program. In the budgeting phase, your program's costs are tied to the rest of the Air Force's monetary needs for coming years and your program is prioritized relative to its importance and the current probability that it will be successfully fielded. *[Refer to HQ USAF/PE Primer, The Planning, Programming, and Budgeting System, HQ USAF/PE.]* In the past several years GAO has noted that virtually ***not one program has received full funding***. As a result, program managers have been forced into the position of having to restructure their programs *"on-the-fly."* Your challenge in today's environment is to structure your software development program to respond readily and aggressively to uncertain funding. [GAO91]

Field Marshall Erwin Rommel defined success on the battlefield as the ability to be flexible and adapt to volatile wartime conditions. He explained that,

Success comes most readily to the commander whose ideas have not been canalized into any one fixed channel, but can develop freely from the conditions around him. [ROMMEL53]

Success in planning for software management is the ability to stick to your plan and also to have a plan flexible enough to adapt to changes in the development environment. The planning process discussed in the next chapter continues throughout the software life cycle and is one of the most crucial activities you must perform as a manager.

REFERENCES

[BENNATAN92] Bennatan, E.M., On Time, Within Budget: Software Project Management Practices and Techniques, QED Publishing Group, Boston, 1992

[BORKY91] Borky, Col John M., communication to SAF/AQK regarding draft AFPAM 63-116, December 11, 1991

[BOURCET75] de Bourcet, GEN Jean, Principles of Mountain Warfare, Oxford University Press, London, 1775

[CUSUMANO95] Cusumano, Michael A., and Richard W. Selby, Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People, The Free Press, New York, 1995

[DEMING86] Deming, W. Edwards, Foreword to M. Walton, The Deming Management Method, Dodd, Mead & Co., New York, 1986

[DEUTCH93] Deutch, John, as quoted by John Moore, "CIM Will Play Key Role in NPR Challenge, Paige Declares," *Federal Computer Week*, September 20, 1993

[DSMC89] *Using Commercial Practices in DoD Acquisition: A Page from Industry's Playbook*, report of the Defense Systems Management College 1988-89 Military Research Fellows, Fort Belvoir, Virginia, 1989

[DSMC90] Caro, Lt Col Israel I., et al., Mission Critical Computer Resources Management Guide, Defense Systems Management College, Fort Belvoir, Virginia, 1990

[DSMC92] "The Acquisition Program Baseline Fact Sheet," *Program Manager's Notebook*, Defense Systems Management College, Fort Belvoir, Virginia, 1992

[EASTERBROOK92] Easterbrook, Gregg, "Stealth-Creators," *The New Republic*, January 6 & 13, 1992

[GAO86] General Accounting Office, *Sergeant York: Concerns about the Army's Accelerated Acquisition Strategy*, Report to the Chairman, Committee on Governmental Affairs, United States Senate, GAO/NSIAD-86-89, May 1986

[GAO91] "Memorandum: Comments on Successful Acquisition of Computer Dominated Systems and Major Software Developments," US Government Accounting Office, Washington, D.C., January 25, 1991

[HUGHES92] Hughes, David, "Digital Automates F-22 Software Development with Comprehensive Computerized Network," *Aviation Week & Space Technology*, February 10, 1992

[HUMPHREY90] Humphrey, Watts S., Managing the Software Process, The SEI Series in Software Engineering, Addison-Wesley Publishing Company, Inc., 1989

[IEEE87] IEEE Standard 1058.101987, *Standard for Software Project Management Plans*, Institute of Electrical and Electronics Engineers, Inc., New York, 1987

[JAYBOUR91] Jaybour, Lt Col W. Jay as quoted by Michael A. Dornheim, "Air Force's Hands-Off Approach Speeded ATF Testing Programs," *Aviation Week & Space Technology*, July 1, 1991

[KEMP94] Kemp, Dan, "CSC Software Development at Syntex: A Case Study," briefing, 1994

[KILE91] Kile, Maj Raymond L, USAFR, *A Process View of Software Estimation*, HQ United States Air Force/C^4 Plans and Policy, Washington, D.C., June 1991

[KINDL92] Kindl, LTC Mark R., Software Quality and Testing: What DoD Can Learn from Commercial Practices, US Army Institute for Research in Management Information, Communications, and Computer Sciences, Georgia Institute of Technology, Atlanta, Georgia, August 31, 1992

[LENIN86] Lenin, Vladimir Ilych, as quoted in *Voyeneo istorichiskiy zhurnal*, March 1986

[MARCINIAK90] Marciniak, John J. and Donald J. Reifer, Software Acquisition Management: Managing the Acquisition of Custom Software Systems, John Wiley & Sons, Inc., New York, 1990

[MRAZ91] Mraz, Stephen J., "Face-off Over Tomorrow's Fighter," *Machine Design*, March 7, 1991

[NORDWALL93] Nordwall, Bruce, "FAA, IBM Share Blame for Automation Delays," *Aviation Week & Space Technology*, March 15, 1993

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[REILY92] Reily, Lucy, "Arms Software Hits Flak: GAO Targets Pentagon on Costs and Scheduling," *Washington Technology*, August 27, 1992

[RICE91] Rice, Secretary Donald B., as quoted by Patricia A. Gilmartin, "US Lawmakers Tighten Scrutiny of B-1B and C-17 Aircraft Programs," *Aviation Week & Space Technology*, March 4, 1991

[ROMMEL53] Rommel, Field Marshall Erwin, B.H. Liddel Hart, ed., The Rommel Papers, Harcourt Brace & Company, New York, 1953

[ROONEY90] Rooney, Thomas R., as quoted in "ATF Avionics Met Dem/Val Goals, Providing Data for Flight Tests," *Aviation Week & Space Technology*, September 24, 1990

[SCHWARZKOPF88] Schwarzkopf, GEN H. Norman, "Food for Thought," *How They Fight*, 1988

[SUMMERS81] Summers, COL Harry G., On Strategy: The Vietnam War in Context, US Army War College, Carlisle Barracks, Pennsylvania, 1981

[SUN500BC] Sun Tzu, Samuel Grifford, ed., The Art of War, Oxford University Press, New York, 1969

[VESSEY84] Vessey, GEN John W., as quoted in the *New York Times*, July 15, 1984

[WORDEN94] Worden, Col Simon P. and Lt Col Jess M. Spanoable, "Management on the Fast Track," *Aerospace America*, November 1994

CHAPTER 13

Contracting for Success

CHAPTER OVERVIEW

Contracting for the last team member, your industry partner, is your next step. In 1958, Dwight D. Eisenhower, 34th US President, made the statement,

> *It is far more important to be able to hit the target than it is to haggle over who makes a weapon or who pulls a trigger.* [EISENHOWER58]

Have things changed in defense acquisition since 1958! Who makes the weapon is one of the most crucial factors in a successful software development—not only to the program manager but to the war fighter who pulls the trigger. With DoD depending on contractors for almost every major weapon, C2, and MIS system development, contractors are the pivotal members of your software acquisition team. There is a direct (and sometimes catastrophic) relationship between software contractor experience, maturity, and capability and program success. This chapter provides guidance on how to choose the right contractor and bring them on board with the right contract that provides the best working relationship between the Government and our industry partners.

The contracting process begins with the preparation of the Request for Proposal (RFP), the basis for proposal evaluation, source selection, and contract award. The heart of the RFP is the Statement of Objectives (SOO), a "what" document. Do not make the mistake of over specifying your software requirements and telling potential offerors "how" to develop or build your software. The whole purpose of source selection is to give industry the opportunity to develop and propose their optimum solution to your requirement.

DoD acquisition reform initiatives state that to improve the DoD procurement process, the Government must make better use of the advanced technologies available in the commercial marketplace. The advantages of buying commercial software are lower cost, faster acquisition time, and more flexible maintenance. You should require the use of commercial-off-the-shelf (COTS) for all those functions that can be fulfilled by commercially available software. You must also require that your COTS comply with established open systems standards.

CHAPTER 13 Contracting for Success

Subcontractors come in all sizes and capabilities with primes relying heavily on them to develop your software. Make sure the prime's proposal has a formal agreement between you and their team that ensures open lines of communication between you and your software developers so all can work together as a cohesive unit. It is essential to require that all prime and subcontractors share a common process to ensure accurate tracking that facilitates integration of all software components into the final system. If major software development is to be accomplished by subcontractors, they should be able to demonstrate the same high standards required of the prime, including a Level 3 or better maturity rating.

Aside from process, the most important commodity you are buying is ***people****. The team with the most qualified, the higher skilled, the greatest experience in your software domain is a ticket on the train to program success. But that success can only be achieved if the company for whom they work is totally committed to the development of quality software and to your program.*

CHAPTER 13
Contracting for Success

TEAM BUILDING: Attacking the Lion

Four brave men who do not know each other will not dare to attack a lion. Four less brave, but knowing each other well, sure of their reliability and consequently for their mutual aid, will attack resolutely.
— Colonel Charles Ardnant du Picq [DuPICQ80]

Large, complex, software-intensive military applications are beyond the intellectual comprehension of any one individual. A single creative developer can produce all the elements required for a simple PC application, but one individual cannot fully understand a large-scale software development, such as an automated air traffic control system, the avionics for a stealth fighter, or a complex C3I network. Nor can one person manage its design, development, integration, and testing without help. These activities require large groups of highly-skilled, experienced professionals.

Software engineering, more than any other engineering discipline, is an ***extremely human endeavor***. Seasoned managers know that software development programs ultimately succeed or fail, not on the sophistication and power of the tools used by their teams — but on the skills and performance of those teams. There are case studies of organizations that assign two equally matched, independent development groups to a software program to minimize the risk of failure. Team A will be given a sophisticated set of automated tools, finishing the task 20% faster and producing better code than Team B. Then the teams switch tools, and Team A will still complete the task 20% faster, with fewer defects than Team B. Experienced managers are aware that this phenomenon reflects the situation throughout the entire software industry. The **teamwork** factor impacts significantly on your management ability to ***attack the lion***. [ALIC92]

CHAPTER 13 Contracting for Success

The **Computer Resource Working Group (CRWG)** is your first step in team building. CRWG members include the implementing agency, the using agency, the supporting agency, and other DoD and Service stakeholders. Other players such as the testers, contracting officers, in-house software developers, and software engineering laboratories are also included. Because most major DoD software developments are built by contractors, this chapter is dedicated to bringing the final team member onboard. This team member is, perhaps, the most crucial, most important, and most difficult player to manage. The **software contractor** you select can literally *"make you or break you."* Figure 13-1 illustrates the composition of a typical software development team.

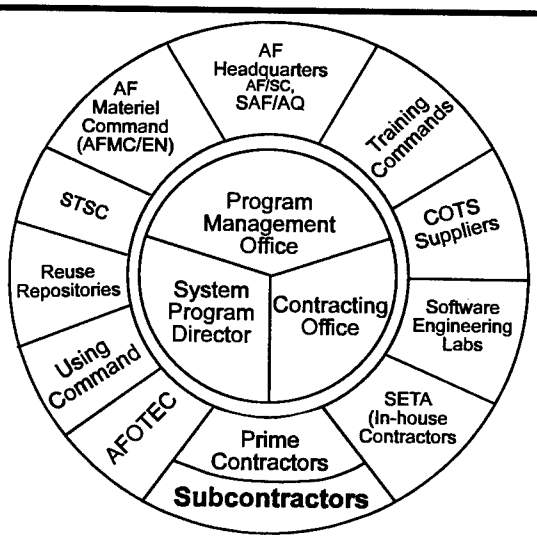


Figure 13-1 Typical Software Acquisition/Development Team

NOTE: See Volume 2, Appendix O, *Additional Volume 1 Addenda*, Chapter 13, *"Contracting for Success."*

Building High-Performance Teams

Recognizing the importance of each team member (contractor and Government) is essential in the accomplishment of your mission. As General George S. Patton, Jr. explained,

> *An Army is a team. It eats, sleeps, lives, and fights as a team. Every man, every department, every unit, is important to the vast scheme of things.* [PATTON44]

Too often government program managers have relied totally on the contract as their management vehicle. Their approach considers problems that arise as the contractor's problem and exerts contractual pressure to solve them. A *"hold-the-contractor's-feet-to-the-fire"* approach has been a common mistake where the ***them-versus-us*** mindset has held the contractor at fault when problems occur. This intractable approach has proven unworkable time and again, because it plays against the teamwork goals of communication, cooperation, mutual respect, and trust. [MARCINIAK90] Mitch Gaylord, US gold medal gymnast in the 1984 Olympics, explained why it is important to share the blame as well as the credit among all team members when he said,

> *A team championship doesn't happen because three people score 10s; it happens because all the guys score well.*

High scoring teams can be defined as groups of individuals who share a common sense of purpose, and are clear about the team's job and why it is important. They envision what the team intends to achieve and develop challenging, mutually agreed upon goals that clearly relate to their vision. Their strategies for achieving team goals are clearly defined, and each team member understands his/her role in realizing those goals.

Although insightful, hands-on management builds cohesive teams, the first step towards building a team that fits the above description is through the **contract**. It represents the best form of communication, a clear statement of the requirement at the start of development, and a framework for establishing a common sense of purpose among government/industry team members. This is where effective team management must start. Keep in mind that the purpose of the development effort is the delivery of an optimum solution — not the exercise of the contract. To deliver a successful solution, successful

teamwork is a must. [MARCINIAK90] Team productivity and morale are enhanced when the contract states clear and attainable goals, provides a mechanism for trust and open communications, and defines a breakdown of tasks and resources that allow the team to function as a cohesive unit. Figure 13-2 illustrates the contracting process.

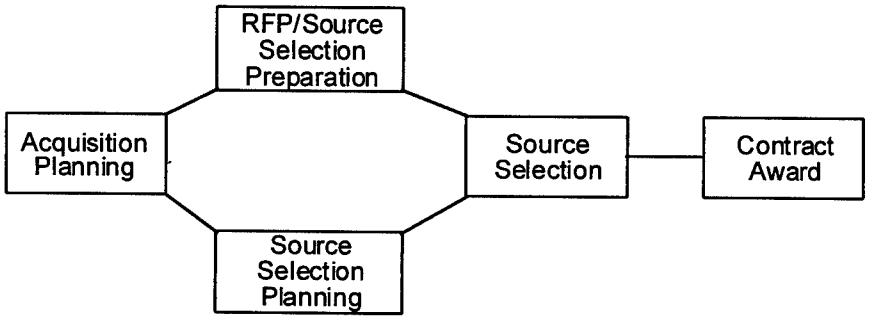


Figure 13-2 Contracting Process

Remember, contractors and their employees are people, too. **Contractors** want to do well, but they also need to earn a profit. When contractors do poorly, it is usually because they have had to cut corners, have otherwise skimped on **process**, or have used lower paid and less skilled employees to lower their costs to submit a winning bid. At the very outset of a competitive acquisition, you must help protect the contractor from these inclinations. The best way is to require that they very clearly (and in detail) describe their software development and management process, the tools they will use, and the skills they will employ. They must also provide explanations and demonstrations that persuade you they can successfully produce your product through their process. You must then evaluate their proposed process with the same care (or perhaps more care) than you apply to the evaluation of their proposed product. When acquisitions fail, government program managers often immediately point their fingers at the hapless contractor; i.e., the contractor is the enemy. Sadly, those program managers do not realize that they, themselves, are the enemy. Do not let this happen to you. Build a team from the outset and develop a common purpose and a productive working relationship with your contractor. ***When programs fail there is no innocent party — both sides are guilty.*** [MOSEMANN94[1]]

J-CALS Teamwork in Action

The first **Joint Computer-aided Acquisition and Logistic Support (J-CALS)** prototype site installation at the Marine Corps Logistics Base, Albany, Georgia is an example *of "teamwork-in-action!"* The *let's-make-it-happen* attitude and cooperative joint government/industry team spirit has delivered marked successes and many joint team firsts. Site installation personnel were faced with what they felt were impossible dates for impossible deliveries of hardware and software developed by the Government and Computer Sciences Corporation. They were convinced there was absolutely no way their prototype was going to hit the streets in time. What was the secret of their success? It was really quite simple once they figured it out. They eliminated the *they*-word, replaced it with the *we*-word, and became the *P-team* (for prototype)! According to LTC Kevin K. Kelley, acting Chief, System Deployment Division, *"It might sound hokey, but don't knock it until you've tried it!"* [KELLEY93]

First, they put a plan together. Government, support contractors, prime contractors, and every stakeholder they could contact provided input. They massaged it, rewrote it, and created a plan that they took to the Office of the Secretary of Defense (OSD). OSD then became part of the *we*-word as they helped staff the plan through all those who needed to sign off on it. Have you ever had a plan approved in less than five days at the OSD level? They did by using the *we*-word.

Next, they expanded the *P-team* to the contracting agency. Their plan got turned into a SOW overnight (literally). The SOW was on contract in 12 working days! Contractor team members were on planes hitting prototype sites two days after contract modification. How did they accomplish so much in such a short time? Smart people kept looking for ways *we-can-succeed*, rather than asking what ***they-had-to-do*** next. The subsequent step was to expand the *P-team* again to include the operational folks at the installation sites. All the things successful teams have to do to get the job done were performed using the *we*-word. These included late nights, 0500-hour runs to airports for new equipment, troubleshooting shorted-out devices, unloading boxes in 100° temperatures, and all the minutia it takes to make things happen where it counts. The *P-team* assembled monthly for in-process reviews where government/industry team members were encouraged to speak their minds. The now extended *P-team* also included site users who were asked to state what they did and did not like about the prototypes, and those issues became action items for *P-team* resolution.

Teamwork made the J-CALS installation an unprecedented success. Facing seemingly insurmountable obstacles, the *we*-approach resulted in prototype site activation 2 days ahead of schedule! Areas needing work are still being uncovered, but the *P-team* is dealing with them. The main thing is that the prototyping is doing just what they wanted it to. They needed to shake out the design, get more interfaces going, and put something tangible into the real users' hands for constructive feedback. The most important factor in this success story was...right, you've got it, the *T*-word — **TEAMWORK**! [KELLEY93]

When contractors and the Government choose to regard themselves as separate teams, trouble lies ahead. In virtually every dispute there is no innocent party. When the Government assumes that shortfalls against expected contract performance are solely the contractor's fault — both parties suffer. If this adversarial situation is allowed to prevail, the Government and the contractor(s) are both losers. Success in these situations, regardless of apparent fault, is salvaged only when both parties are willing to commit to genuine compromise.

CONTRACT TYPE SELECTION

The degree of interaction between you and your contractor depends on the nature of the development effort and the type of contract used. When software requirements are well-defined (such as upgrading or enhancing an existing system) and the risk of development is low, a **fixed-price type contract** is probably the right choice. Where requirements are ill-defined and development risk is high, a more flexible **cost-reimbursable contract** may be more appropriate.

A way to mitigate high technology risk is to consider **multiple award contracts,** or contracts with a provision for **event-driven task orders**. This contracting strategy allows for a closer working relationship, and provides a better chance for arriving at a satisfactory solution because the risk is shared among government and industry team members. [MARCINIAK90] *[FAR, Part 16 provides basic principles and policy guidelines for contract type selection.]*

The same factors influencing cost uncertainty also influence schedule uncertainty. However, there is no equivalent acquisition practice for establishing flexibility in the **program baseline schedule**. Although baseline schedules are often driven by the **initial operational capability (IOC)** need, the IOC is frequently established before assessments of technical risk and relative uncertainty, requirements

definition, design solution, and effort (i.e., schedule) have been made. This is particularly true for unprecedented systems. Program level schedules are, too often, success oriented, do not reflect past actual schedules, and are, in fact, unachievable. Offerors have no real choice during competitive source selection but to respond to these schedules, even though meeting them represents inordinately high risk. As discussed in Chapter 12, *Planning for Success*, ***success-oriented schedules are seldom successfully achieved.*** Schedule pressure often compromises sound engineering and management practices — increasing the risk of poor product quality. One way to deal with unrealistic schedules is to allow partial delivery of the required functionality, with additional functionality delivered in later phases (i.e., incremental development, discussed in Chapter 4, *Engineering Software-Intensive Systems*).

Another solution is to include the concept of **schedule-plus** in your RFP as an approach for establishing program schedule baselines. The schedule-plus approach is selected for the same reasons the cost-plus approach is used, where technology and other factors have sufficient uncertainty and risk. This is equivalent to setting up a management reserve for cost, extended to schedule. It is also warranted if you are planning to use a cost-plus approach, if similar past programs have had schedule performance problems, if a comprehensive Dem/Val phase has not been accomplished, or if you face significant challenges in the areas of performance and/or supportability requirements.

A **schedule-plus contract** includes a baseline (minimum) schedule plus a delta range. This is determined by comparing your proposed schedule to past actual schedules achieved on programs similar to yours. Your schedule must then be evaluated and adjusted to within the delta, as necessary, at program milestones. Schedules adjusted beyond the delta range will require re-approval of the baseline schedule estimate. You can include incentives for the contractor to stay within the planned schedule delta, such as a profit share line tied to the schedule, completion fees, and award fees. Your RFP must also identify specific schedule review elements to determine schedule status. These reviews are based on measurable events and actual data (metrics). The schedule-plus approach is also integrated with the systems engineering master schedule and plan. Firm performance requirements are clearly differentiated from goals, allowing offerors to bid to realistic schedules. Your schedule-plus approach is also integrated into your **C/SCSC system** *[discussed in Chapter 10, Software Tools]*. For example, a periodic schedule **estimate at**

completion (EAC) is accomplished along with the cost EAC. [SPAT92] Ideally, you will not state any specific schedule requirement in your RFP, but allow offerors to propose a realistic schedule which you will evaluate for realism during source selection.

DEVELOPING THE RFP

The contract vehicle must be designed to clearly express a vision of final product goals and development effort requirements. Thus, the development of the **Request for Proposal (RFP)** is your first step towards bringing Government and industry together as a cohesive, high-performance team. The RFP also marks the culmination of the strategic planning process and represents the formal means for communicating government requirements to industry. Too often, the RFP is viewed as an administrative (rather than a technical) document. Its administrative function must be secondary to its technical function. The RFP must contain clear and sufficient technical guidance so the contractor has a definite picture of how the system is envisioned to perform once delivered. It is also important that a technical functional description of software requirements is included and that those requirements are clearly scoped. Inconsistencies, insufficient detail, and inappropriate software requirements will result in an inadequate response from industry to government needs.

NOTE: Because every program has unique requirements, it is beyond the scope of these Guidelines to provide specific information on what is important for every source selection. However, if you consider the suggestions listed herein, you will be well on your way to a successful acquisition.

Considerable time and effort is required to form a comprehensive software development RFP. Your program office ***must work with the future user of the system*** (your customer) to establish requirements, expectations, schedules, and support needs. Both the program office and the user must remember that well-defined, performance specifications yield good contracts and the better scoped the requirement — the better response from industry. [MARCINIAK90]

Early **industry involvement** in acquisition planning often improves the quality of the RFP and fosters a sense of government/industry partnership. Early involvement is an iterative and interactive

communication process. You are required to integrate early industry involvement in your acquisition planning for all competitive procurements that are estimated to exceed $24 million. But, as stated in AFMCP 64-102, *AFMC Request for Proposal Process Guide,* early communications with potential sources on any intended acquisition is plain common sense. Figure 13-3 illustrates the major RFP preparation process components.

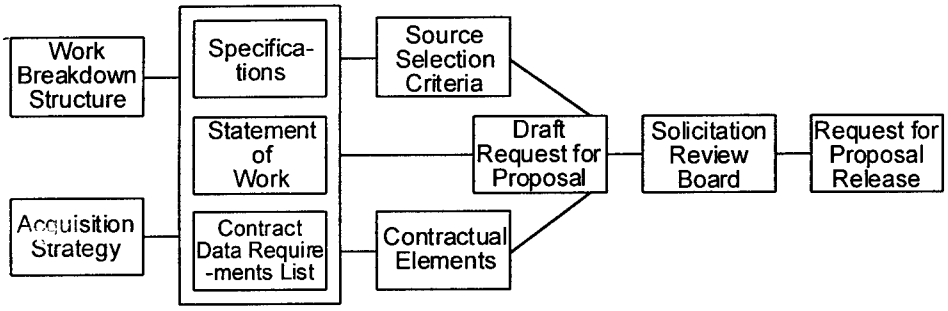


Figure 13-3 RFP Preparation Process

At the end of the following chapters you will find discussions on the software engineering subjects that must be addressed in the RFP. In addition, see

- Chapter 6: "*Addressing Risk in the RFP,*"
- Chapter 7: "*Addressing Software Development Maturity in the RFP,*"
- Chapter 8: "*Addressing Measurement and Metrics in the RFP,*"
- Chapter 9: "*Addressing Reuse in the RFP,*"
- Chapter 10: "*Addressing Tools in the RFP,*"
- Chapter 11: "*Addressing Software Support in the RFP,*" and
- Volume 2, Appendix M, *Software Source Selection.*

RFP Development Team Building

Your acquisition team is your most important resource. It takes highly qualified, inhouse personnel to develop the RFP. Competent people are also indispensable to judge product deliveries and keep the program headed for success once the contract is awarded. Assembling a qualified **acquisition team** to bring on the last team member is another opportunity to practice your team building skills. Your acquisition team must include personnel from the supporting and

using agencies, as well as software, contracting, and cost analysis experts. Team members creating the portion of the **Statement of Objectives (SOO)** related to software development must be knowledgeable in software engineering and management. Program office personnel serve as team leads and must make sure the user and supporting organizations' needs, concerns, and desires are fully addressed.

The **RFP team,** the core of the **source selection (acquisition) team,** develops the technical, managerial, and cost requirements and evaluation criteria for the acquisition. The software specification must be thorough and **source selection evaluation criteria** rigorous. Your evaluation criteria should only, however, measure those items that are valid discriminators and directly traceable to requirements. The RFP must require that each proposal submitted contains sufficient information for a thorough assessment of each offeror's software development and Ada experience, tool availability, product assurance, team skills and experience, software support, and program management capabilities. The offeror's proposal should describe how their product and process will satisfy required and desired functionality. *[See FAR 15.6 and AFFARS, Appendix AA and BB.]* The RFP should include:

- Clear, concise statements of specific tasks (quantifiable, measurable, and testable),
- Specifications and standards tailored to program needs (relying on commercial standards and practices, whenever possible),
- Planned use of government-furnished equipment (GFE), government-furnished information (GFI), and/or government-furnished software (GFS),
- Requirements for the contractor to provide a comprehensive layout of program schedules (internal reviews, formal peer inspections, testing, technical meetings, etc.),
- Requirements for relevant and pertinent domain experience,
- Requirements for a thorough Software Development Plan (SDP) *[discussed in Chapter 14, Managing Software Development]* and plans for its implementation and updates, to include a proposed test process plan,
- Requirements to describe use of Ada, Ada SEE's, COTS, and reuse,
- Requirements for appropriate software documentation,
- Requirements for an open systems architecture and architecture performance verification,
- Requirements for resources growth/margin verification,
- Requirements for a total life cycle/total systems perspective,

- Requirements for prototyping and/or demonstrations (ideally, a demonstration of an executable architecture as part of the proposal),
- Requirements for a progress, process, and quality measurement program, including a specific Metrics Usage Plan,
- Requirements for a software quality assurance (SQA) program,
- Requirements for supportability planning,
- Requirements for a Risk Management Plan and its implementation,
- Requirements for a Process Improvement Plan and its implementation,
- Requirements for a process control mechanism [e.g., Process Weaver® *(discussed in Chapter 10, Software Tools)* or equivalent],
- Requirements for developing interface software with other system software and/or hardware,
- Requirement for assessing software development maturity/ capability,
- Planned communications with any IV&V contractors or agencies, and
- Requirements for delivery of the life cycle support environment. [DSMC90] [MICOM91]

NOTE: The first item "*clear, concise statement of specific tasks*" is particularly important. For example, a requirement to respond to user requests in "*real-time*" is ambiguous because there is currently no standard definition of the term "*real-time.*" It is much better to provide a numerical value (such as "*within one microsecond*") for such a requirement.

Statement of Work (SOW)

The **SOW** is the primary document for translating performance requirements into contractual tasks. The SOW must be consistent with the **summary WBS** and contain sufficient information for the offeror to prepare a detailed **contract** WBS *[discussed in Chapter 12, Planning for Success].* It must also contain tasking information for **contract data requirements lists (CDRLs)**. While the SOW states the specific tasks to be performed, ***it must not tell the offeror how to do the required work***. Do not spell out specific qualitative and quantitative technical requirements in the SOW. Instead, offerors must be solicited to propose their solution to your stated need

As **Salvucci** explains, it may no longer be appropriate to rely on the SOW as a guide for contractors' efforts. In the article, "*Do We Need*

the Statement of Work?" Lake also proposes that a lengthy SOW with detailed taskings is inappropriate if the RFP has a specification, a WBS, CDRLs, and requires a SEMP (with accompanying SEMS). [LAKE94]

Contractual Data Requirements List

The RFP's **contract data requirements list (CDRL)** is the primary vehicle for acquiring documentation from contractors. It lists all **data item descriptions (DIDs)** that apply to the development program. DIDs describe the data the contractor is required to provide, along with delivery instructions (such as media or format). Each CDRL entry contains the DID identifier, title, requesting organization, distribution, referenced SOW task(s), and a remarks section where information (such as tailoring) is included. ***Nearly all DIDs require tailoring for appropriate application to a contract.***

Subcontracting

On major DoD software-intensive system acquisitions, seldom is all the work performed by one contractor. Instead, the winning contractor is the **prime** who in turn arranges with other contractors, **subcontractors**, to complete some portion of the effort. Currently, ***over 60% of total weapon systems development and production efforts are subcontracted.*** [BAKER92] This includes most of the software for these systems. These parallel hardware/software efforts have resulted in adversarial relationships and turf battles between the prime (hardware) and the subcontractor (software)—often impeding software development efforts. Although the prime is responsible for product quality, the prime must include in their subcontracts all contractual requirements necessary to ensure that software products are developed in accordance with prime contract requirements. It also states that the Government must be given access to subcontractor facilities to review software products and activities required by the contract.

Inappropriate contract types are often a subcontractor problem because software developers seldom have solid requirements. A **firm-fixed-price (FFP) contract** is risky because the subcontractor views all requirements changes as *out-of-scope* changes, whereas the prime views them as *in-scope* with the **System/Segment Specification (SSS)**. Since the subcontractor must develop the **Software Requirements Specification (SRS)** from the SSS, many problems occur when the SRS does not accurately reflect the performance requirements desired by the Government. Other problems occur

requirements desired by the Government. Other problems occur when too many specifications are left up to the discretion of the subcontractor. These contrasting views of responsibility are illustrated in Figure 13-4. When software programs experience difficulties, primes and subcontractors often resort to blaming each other. When schedules are not met, it is always the other company's fault!

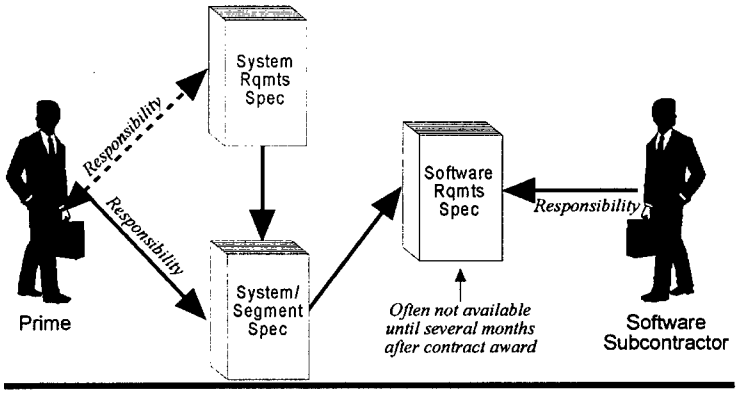


Figure 13-4 Prime/Software Subcontractor Development Responsibility [MOORE]

In addition to differing development effort views, primes and subcontractors frequently have differing business perspectives and goals. Software subcontractors are often hired only for the development effort, whereas the prime makes a profit throughout the life cycle up to deployment, as illustrated in Figure 13-5 (below). [MOORE]

Subcontracting can also be enigmatic because DoD managers have no direct control over subcontractors. You can only direct and manage your prime contractor. It is up to the prime to direct and manage their subcontractors. Figure 13-6 (below) illustrates the communications distance between you and your software developer and between the user and the software developer. In the RFP, you must ensure that the prime contractor provides the needed direction the Government requires of the software developer. One approach is to suggest that the prime contractor develop and follow a **Subcontractor Management Plan** that ensures government visibility and participation in the management process. After contract award, the program office, with the consent of the prime contractor, should be allowed to make periodic escorted visits to critical subcontractors. The plan should also make provisions for a joint team (user/SPO/ prime/subcontractor) software development effort.

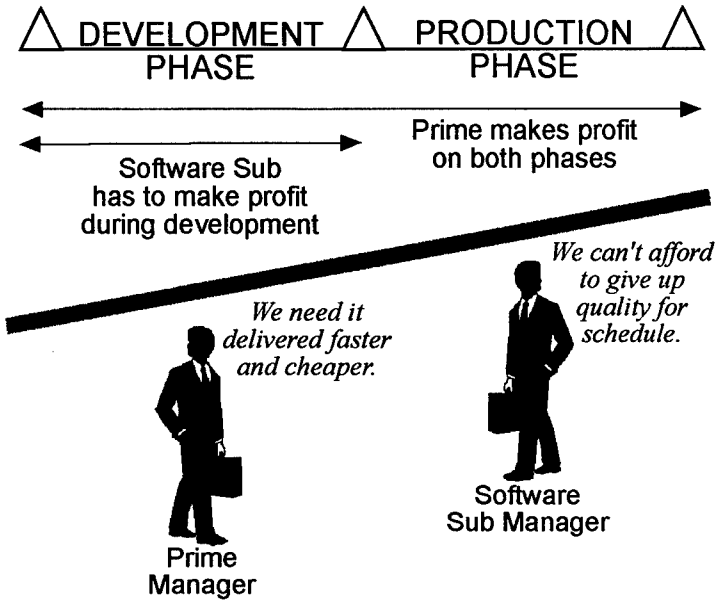


Figure 13-5 Prime-Sub Different Business Perspectives [MOORE]

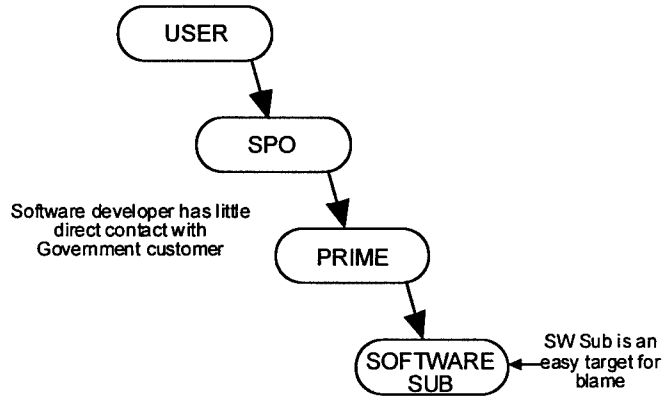


Figure 13-6 Chain of Government-Subcontractor Communications [MOORE]

Another way to control subcontractors is to ***"suggest"*** in your RFP that they follow standard provisions and that proposals will be evaluated accordingly. During source selection, your contracting officer can also require that subcontractors submit proposals along with the one submitted by the prime. *[Software Capability Evaluation*

(SCE) requirements concerning subcontractor management are found in Volume 2, Appendix M.]

Joint Venture Partnerships

To avoid difficulties associated with subcontracting, some firms enter into **joint ventures**. In a joint venture two or more prime contractors create a single corporate entity for a specific program. A recent example is the Boeing-Lockheed-General Dynamics joint venture for developing the **F-22**. *[NOTE: It is now Boeing-Lockheed due to recent corporate mergers.]* This arrangement has provided a low-risk business approach for all joint team members and the Government. The contractual agreement among the three companies calls for equal sharing of any cost overruns. If one team member encounters a problem, the other two members must help solve it — or they all suffer. The degree to which each company has subordinated self-interests to achieve team goals for the F-22 is another first in defense contracting lore. [VELOCCI91]

Joint ventures place much of the management burden on the contractors. Because the joint venture establishes the team members as equal team partners, it enhances cooperation and avoids many problems associated with subcontracting. If one member is lax, the other member(s) can exercise considerable leverage. Joint venture also removes much of the technology transfer requirements from the government program office. When evaluating joint venture proposals, source selection officials must ascertain whether a joint venture team is a viable competitor. However, the contractual arrangement details between the joint venture team members are not the Government's concern. [KRATZ84]

While joint venture partnerships eliminate some subcontracting issues, they present other management challenges associated with contractor teaming arrangements. This complex agreement between companies requires the formation of a new corporate entity, election of officers and a board of directors, assignment of personnel, and establishment of accounting and administrative procedures. Large responsibility is placed on parent members and on what is, in essence, a start-up company. Therefore, joint ventures should not be construed as the end-all panacea for subcontracting quandaries.

The subcontracting dilemma may not be altogether eliminated either, since joint venture members may not be precluded from subcontracting out a portion of their individual tasks, of which software will always

be a target. This can place an additional management layer between you and your software developer. Another potential disadvantage to joint ventures is that while equal partners are established, they may be geographically separated. This leaves you with three (or more) entities with which to interface at multiple locations, further complicating program management. [KRATZ84]

SPECIAL SOFTWARE RFP CONSIDERATIONS

As you learned in Chapter 2, *DoD Software Acquisition Environment*, new acquisition streamlining initiatives state that essential government needs should be met with a minimum SOW. Mills expands on the concept of ***teamwork*** to one of government/industry "***partnership***," where both team members share the responsibility for success. By transforming the Government's contract monitoring role from the older documentation-driven, review it, approve it, and baseline it paradigm, the new partnership role places the contractor solidly in charge of the process and the emerging product.

To meet DoD's requirement for an overall manageable procurement, five key elements are essential in a RFP for a major software-intensive system. A risk management path for each element entails a minimum SOW with reduced proposal instructions. Offerors are then left with defining their actual approach. There is a strong synergistic relationship among the software RFP elements. For example, by allowing contractor control of baselines until very late in development, without the need for 100% correctness and completeness of documentation, it is possible to reduce former resistance to open sharing of technical information. ***This promotes partnership***. At the same time, it is essential to clearly agree upon technical milestone points and formal reviews, and to establish a framework to ensure contract progress is achieved. Confidence in the offeror's software development process permits greater trust in the offerors ability to achieve contract milestones. The five key software RFP elements are:

- Software development process,
- Contractor documentation and formats,
- Contractor control of baselines,
- Direct technical visibility, and
- Proactive risk management *[discussed in detail in Chapter 6, Risk Management]*.

Software development process. As discussed in Chapter 7, *Software Development Maturity*, offerors must be pre-qualified at a maturity Level 3 and required to employ systematic, well-documented software engineering practices which complement your program's risk management strategy. A mature contractor process helps ensure that they will produce supportable, quality software on schedule in a predictable, consistent manner. The contractor's practices must also be documented, maintained current by the development team, and be available for Government review. This supports the need for continuous verification of process maturity and effectiveness.

Contractor documentation and formats. Documentation deliverables should maximize the use of information in the form and format used to develop the software.

Contractor control of baselines. By allowing the contractor to retain configuration and engineering control of baselines until they are stable, frees the developer from the government review and approval cycle which also supports partnering.

Direct technical visibility. This may be implemented with the following requirements:

- The contractor must plan and implement a means for sharing software development information with the Government. The contractor should be required to provide access to current working documentation in the language and format normally used for software development. This includes Government access to software engineering tools and databases.
- Documentation, where possible, should reside in electronic format in the automated software engineering environment.
- The contractor must plan the information sharing mechanism so that little or no contractor assistance is required for government personnel to access information. The information can be used as a basis for formal government recommendations to the contractor, and whenever practical, should be used to simplify the formal technical review process. Thus, you need not provide formal approval of shared information on a day-to-day basis.

Proactive risk management. Risk was reduced by requiring the delivery of a series of documents. Each deliverable was typically reviewed and approved by the Government to ensure quality and to independently verify contractor adherence to schedule. In principle, this document-driven contract monitoring was an efficient way to

manage software development risk and perform program oversight. In practice, the oversight role progressively removed the developer from responsibility for design as each new document was approved. Since the Government performed the review and found the errors, the contractor only had to deliver a product on schedule and correct any errors the Government found. This approach too often led to increased reliance on testing and diffused the responsibility for quality problems, which often remained hidden until system delivery.

Providing Government access to software development information promotes partnership by removing disincentives for information sharing. Technical concerns are, for the most part, transferred from the schedule enforcement aspects of oversight. The monitoring role includes a technical function where you participate in reviewing emerging products and gain visibility into program progress and product quality. Knowing the you are monitoring product quality prior to delivery, the contractor is more inclined to build quality into your product. You also have the opportunity to directly verify process effectiveness and program metrics.

To ensure industry participation in contract monitoring, your RFP should request that offerors address the important aspects of program management and monitoring in their proposals. These requirements must be brief and concise in the SOW, thus, requiring expansion and clarification in the offeror's proposal. The offeror's commitment to partnership and an improved contract monitoring role should be an important software source selection criterion *[discussed below]*. This not only provides a performance incentive, it rewards offerors who are committed to quality and process improvement. This monitoring role must also be strongly supported by proactive risk management.

When preparing your RFP, it is often difficult to communicate the need for a comprehensive response from industry. Because software is just one part of a system, the detailed proposal instructions necessary to obtained a risk-based proposal can easily be confused with non-value added requirements. When upper management is told that source selection requires more time and attention, your program schedule can often become the constraining factor. Despite these obstacles, ***reducing software risk during source selection is one of the most crucial management activities you will perform.*** Unfortunately, the acquisition planning and RFP preparation process is lengthy and challenging. We must do more with less and we are all anxious for a stable program which can be smoothly budgeted. Schedule and availability of program funds are often the our main

constraints. However, at the heart of the challenge is the requirement to meet your particular program's needs by selecting the contractor of whom you are assured represents the least risk with the greatest potential for success. [MILLS95] *[See "Selecting the Last Team Member" below for a list to what to look for in a winner.]*

Commercial-off-the-Shelf (COTS) Software

As explained in Chapter 2, *DoD Software Acquisition Environment*, our procurement cycle for major software-intensive systems is usually 10-plus years from inception to IOC. This acquisition cycle contrasts with the commercial software life cycle in which a product is enhanced in 6-9 months, a new product is developed in 12-18 months, and a product is obsolete in 36-48 months. [FAA94] ***Hence, DoD is procuring systems which are technologically obsolete by the time they are fielded.*** Strict regulations and the long acquisition cycle translate into commensurately exorbitant costs for custom developed software. Software costs are also high because DoD has borne the entire financial burden for software maintenance (either contracted or in-house).

COTS Advantage

The Pentagon's long-range budget plans show future funding will be devoted to **technological modernization** after several years of steep declines. As **General John Shalikashvilli**, Chairman of the Joint Chiefs of Staff, proclaims, "*modernization is the key to future readiness*" of US military forces. [SHALIKASHVILLI94] By investing in the development and procurement of new, advanced technologies, America's 21st century military will continue to be equipped with the ***technological advantage*** — the hallmark of US strategic military doctrine since World War II. [MORROCCO94]

To modernize our software technology, cut costs, increase quality, and indeed, to maintain our superpower status, ***we must become software users, instead of custom software developers.*** By removing requirements for government-unique accounting standards, product specifications, and processes, DoD's purchasing system must become more compatible with that of the commercial marketplace. In addition, preference for the use of commercial standards and processes (established through the June 1994 **Perry Memo**), protection of technical data rights for commercial items, and a broadened exemption from cost data requirements is essential. [SULLIVAN94]

As discussed in Chapter 1, *Software Acquisition Overview*, the 1994 report of the **Defense Science Board** identified an urgent need to integrate major parts of our defense-industrial base with our commercial-industrial base. It concluded that this integration allows DoD access to those technologies, products, and processes dominated by the commercial sector that are far more advanced than military technology (e.g., electronics, software, computer hardware, robotics, telecommunications, etc.). [HERMANN94] In an interview with *Government Computer News*, **Mosemann** explained that increased reliance on COTS technology will allow DoD to buy top-of-the-line new products. By buying commercially competitive products, *"We're looking at getting as close to a Cadillac as we can get at a Chevrolet price,"* he quipped. [MOSEMANN93] The advantages to purchasing COTS software include:

- Intense competition leading to commodity-like pricing and alternative sourcing,
- Hundreds, to millions, of product users leading to low defect latency,
- Market pressures to rapidly innovate, leading to better products,
- Some degree of standardization leading to interoperable components between otherwise competing software manufacturers,
- Ease of migration to often revolutionary, future technologies,
- Built-in compliance with standards (although these are often as *de facto* as they are *de jour*),
- Exploitation of lower cost, quicker evolutionary development processes, and
- The use of commercially-developed tools and software engineering environments for increased automated development. [FAA94]

By exploiting these advantages in our acquisitions, we can achieve lower costs, faster developments, and more flexible maintenance with the ability to phase in new requirements throughout the software life cycle. We can gain greater capability by using COTS [and GOTS and NDI (i.e., reuse)] instead of relying solely on new developments to meet our needs. COTS increase productivity by decreasing the lines-of-code to be developed, and improve quality by using code that is already tested and proven. In fact, COTS products can sometimes entirely preclude development, because they are often cheaper and more readily available than developed software. A well-used COTS application has been refined through updates (or versions) and corrected for latent defects — making it more reliable than newly developed, untried code. Additionally, vendor support of COTS is usually available. In the DoD environment, obvious applications

such as word processors, spreadsheets, and cost models should always be acquired as COTS. In fact, ***COTS software should be purchased for all software requirements that can be fulfilled by commercially available products.***

NOTE: See Volume 2, Appendix O, Additional Volume 1 Addenda, Chapter 11, Addendum B, "*Electronic Combat Model Re-engineering,*" and especially the sections: *"COTS Software and Ada;" "The Benefits of Ada;"* and *"Ada and COTS."*

COTS Integration

As you learned in Chapter 2, *DoD Software Acquisition Environment*, the **Technical Architecture Framework for Information Management (TAFIM)** is commonly known as a standards-based architecture (SBA). The integration of COTS with an SBA involves the partitioning of system capabilities into well-bound modules. An example of this integration is the mapping of modules from the TAFIM to traditional client/server components (presentation management, application function, data management.) Encapsulation refers to the classification and isolation of each system capability into appropriate client/server components. Further encapsulation within a component is sometimes necessary to ensure greater flexibility and ability to interchange COTS with other system components. This controls maintenance costs by allowing the developer to incorporate new technology rapidly with minimal impact to the other partitioned components.

Shrink-wrapped COTS products require little more than installation and configuration management. The use of a development language is not required to integrate software capability. Some shrink-wrapped examples include:

- Embedded communication software that contains all the logic necessary to enable the communications function, with only connectivity and configuration parameters required.
- Relational database management systems, from simple personal computer products to distributed solutions that provide centralized access to one or more disparate data sources. Some of these data access products are referred to as *middleware*.
- Vendor applications that perform standard commercial functions required by DoD, such as financial data tracking and administration, which provide common industry calculations and formula

- Advanced industry scanning and image manipulation technology, such as Electronic Data Interchange (EDI) translators and wireless communications products.

Advantages of using shrink-wrap COTS are the incorporation of the latest technologies, automating manual processes, low initial cost relative to a new development effort, reduction of maintenance costs, and timely solutions to changing requirements. Often called the *"golden handcuffs,"* the disadvantages of shrink-wrap COTS include: proprietary development language (i.e., 4GLs), difficult customization to unique DoD requirements, inconsistent support for established standards, lack of real-time support, limited support for a centralized DoD data repository, and changing vendor relationships and can result in tight integration between two products for one version and loose (or no) integration for another version. Hillier explains, the keys to removing the "*golden handcuffs*" of reliance on vendor proprietary solutions include:

- Implementing the application component in a standard 3GL, independent of any proprietary COTS solution;
- Limiting the use of *defacto* vendor standards and seamless integrated COTS solutions to environments with no migration or cross functional requirements; and
- Selection of "*shrink-wrapped*" COTS and development environment COTS based upon product superiority within each selected partitioned component.

Two main middleware technologies occupy the center stage for future *de facto* architectures: **Remote Procedure Call (RPC),** and **Object Request Brokers (ORB).** Each have their strengths and weaknesses and the robustness and maturity of these vary significantly. RPC technology is the most mature, and perhaps the least innovative. It can also be used as a building block for other technologies, such as ORBs, object databases, distributed file access, and distributed transaction processing. RPC implementations, especially the **Distributed Computing Environment (DCE),** are readily available from vendors (i.e., corporate members of the Open Software Foundation [OSF]).

Increasing attention is being paid to the **Common Object Request Broker Architecture (CORBA),** discussed in Chapter 10, *Software Tools*. This ORB provides the mechanism by which objects *seamlessly* request and receive information and provides interoperability among applications in a heterogeneous distributed environment and

interconnections among multiple object systems. Objects implemented on other platforms can be accessed regardless of the programming language used, thereby providing a multi-language class library. For instance, Ada 95 could access C++ objects in one implementation, Smalltalk objects in another, Eiffel objects in still another implementation. Vendor products are emerging that pre-compile **CORBA Interface Design Language (IDL)** into 3GL source code. While neither DCE nor CORBA are considered to be dominant, near-term product availability favors DCE, while longer term solutions favor CORBA (possibly DCE-based). [HILLIER95]

Factors to consider when selecting COTS products include:

- **Security**. Does the COTS solution provide two-way authentication, authorization, access control, privacy, and integrity?
- **Middleware supplier independence**. Does the solution truly provide greater independence from computer and network suppliers or does it simply shift to different ones?
- **Server supplier independence**. Does the solution employ general purpose server- independent middleware, to reduce vendor reliance and avoid affecting the client interface to the middleware layer?
- **Standards and interoperability certification**. Does the vendor supply verification of standards compliance (when appropriate)?
- **Training curve**. Does the product require considerable training and does the vendor provide it?
- **Costs**.
  - **Commodity pricing**. Is the COTS based on plug-compatible standards to promote lower licensing costs?
  - **COTS product and perquisite support products**. Are there any hidden costs?
  - **New requirements**. How mutable is the COTS product to changing requirements? Do simple to moderate changes induce a ripple effect across the rest of the software?
  - **COTS version upgrades**. Are major changes planned in the COTS product to maintain a competitive edge with rivals?
- **Performance**. Does the COTS product provide (or enable the development of) the capabilities needed to satisfy mission needs?
- **Time to develop**. Will the use of the COTS product decrease (or at least not extend) the time to market for the system?
- **Memory, storage, and processing power**. Processing capability is cheap, but some solutions take lots and lots of memory which eventually increases cost at the PC and work station level.

With what degree of efficiency does the software suite employ memory with reserves for future growth? For COTS, NDI, and software reuse, criteria should assess the complexity of integration. How transportable is the software and standard hardware architecture?

The level of COTS integration achievable depends on the amount of COTS software that can meet mission needs, of which there have been 100% COTS solutions. This situation, however, is not the norm. Most military systems require the addition of mission-specific logic. Some even have unique performance requirements that almost entirely preclude the use of COTS. These classes of systems can be categorized as:

- Predominantly COTS,
- Integration of COTS, new 3GL, and reusable 3GL assets, and
- Predominantly 3GL.

CAUTION! **COTS products derive their quality (identification of latent defects) by an extensive body of users who identify the defects for fix in subsequent releases over a prolonged period of time. Quality is not necessarily designed-in as would be the case in a Cleanroom-based Ada development.** *[See Chapter 15, Managing Process Improvement, for a discussion on Cleanroom.]* **Accordingly, where safety of life or other situation would suffer if the COTS contains hidden bugs, custom development may be preferable.**

COTS Integration with Ada

Hillier explains that integration of COTS with Ada denotes any interface between a COTS product and Ada code (directly, through another language, or through another COTS product) which require a developer, maintainer, or migrator to integrate the COTS and the Ada code.

The predominant interface solution is a **binding** or **direct call**. Bindings provide the COTS interface for 3GLs not directly supported in a vendor product, a necessary component in the integration process. Many COTS products provide C interfaces as part of a standard or extended shrink-wrap product, while various Ada

repositories, Ada tool vendors, and some COTS vendors provide Ada interfaces or bindings. Vendor bindings are normally proprietary. In other words, they are unique to the vendor's implementation, except in the instance of some standards-compliant interfaces, such as X-Windows.

Abstraction levels are fundamentally achieved through *thick* and *thin* bindings. Thin bindings normally implement a one-to-one, low-level interface providing direct access to COTS functions. This solution has a low overhead in terms of time-critical performance. However, a one-to-one mapping to a vendor solution still ties the code to the vendor product. Thick bindings, on the other hand, provide a higher level of abstraction to support portability and reduce complexity.

If a developer decides to implement a thick binding for a particular COTS interface and the vendor no longer supports the product, then the developer/maintainer has the capability to port to another environment more easily. Thick bindings often are composed of lower level thin bindings with the addition of higher level interface packages, and allow for portability, flexibility, and reusability of the software assets employing them. The most frequent need for interfaces to COTS products are for user interfaces, databases, operating systems, and data communications. Several bindings have been created for each, though some standard bindings are emerging through a standardization process or by Government tasking.

COTS Integration Lessons-Learned

The following are lessons-learned on the Loral Service Layer ReARC COTS software integration program:

- The customer's willingness to pay to gain the benefits of COTS must be understood. There must be a willingness to: abandon/ modify some requirements if COTS cannot meet them; deal with sometimes disparate user interfaces; and accept the lack of control that COTS brings to the change process (slower reaction to problems, less desire to incorporate changes).
- Metrics for COTS integration are difficult to extract. *[The high degree of integration between COTS and ReARC developed code made correct allocation of efforts difficult to determine. Late selection of COTS product baseline (and even which requirements would be met with COTS) made the allocation of funds difficult between COTS and developed code.]*

- While COTS are less expensive than developed code for large, general applications (e.g., operating systems, DBMS), experience does not indicate this is true for smaller, niche products. Savings are anticipated in FQT and O&M. However, the customer is not always willing to be flexible enough to achieve reduced costs from COTS.
- Non-technical COTS selection criteria are as important as the technical. These include: vendor stability; product cycle; availability of support for back-level releases; and willingness of the vendor to work with the contractor.
- Managing product licenses can be a big headache. Therefore, get as flexible a set of licensing terms as possible — a site license is the best; *node-locked* licenses are the worst. Remember that every hardware baseline change can impact your COTS product baseline causing days of down time in the development lab.
- Do not believe what you read — *fly before you buy*! Early prototyping and integration with developed code (before CDR) is the only way to ensure that the product performs as advertised.
- Understand prerequisites and dependencies between products, especially when planning upgrades. *[ReARC is planning a later upgrade than desired due to the dependency of other COTS products.]*
- Make sure procurement processes are in place to expedite COTS delivery.
- Establish hardware and COTS product baselines early. *[ReARC changes from a heterogeneous hardware environment (Sun and RISC) to a RISC-only environment was positive, but came too late and drove up costs by having to find replacement COTS products. Slow response from vendors can best be dealt with if problems with COTS are found early.]*
- Work with the customer to negotiate modifications to documentation requirements for COTS products.
- The cost of integrating COTS is much more front-end loaded than the cost of developing code. *[The developed code algorithm used on ReARC allocated 50% of cost to PDR and CDR phases, and 50% to Code and Unit Test, Software Integration, and CSCI Testing. They estimated that 75% of COTS integration costs come in the PDR and CDR phases. The potential exists for substantial savings in FQT, if agreements regarding verification of requirements satisfied by COTS are established early.]*
- Spend as little time as possible on paper trade studies. Get products to the lab for prototyping and integration. [RAND95]

More Cautions About COTS

The advantages of COTS (availability, cost, reliability) are evident in that all major DoD software-intensive systems are progressively increasing the use of COTS hardware and software. Aside from the advantages, there are, however, also some downsides to the use of COTS. Most COTS software is proprietary and the supporting agency cannot make changes to it. Therefore, COTS is not a good choice for weapon systems where the software must be continuously enhanced in response to changing mission requirements. Also, as you learned from the **Navy Seawolf BSY-2** program in Chapter 5, *Ada: the Enabling Technology*, sometimes it is quicker and cheaper to simply build in Ada. You may be hard pressed to meet highly demanding volume and reliability requirements with commercial products not designed to perform under large-scale military conditions.

There are some fundamental differences between commercial software products and developmental software, as the final report of the **1991 Joint Command COTS Supportability Working Group** concluded. [COTS91] Although cheaper than developing it yourself, be aware it is often difficult to integrate all the COTS applications (especially for weapons systems) needed to provide the required functionality. Even if your integration is successful, (for example, with 26% COTS combined with 74% developmental software) you can encounter configuration control problems. With new versions of each vendor's product being delivered at varying times, taking full advantage of the enhancements of each new product through changes in interfaces and interoperability with existing software can be like trying to catch a leprechaun to get his pot of gold. Every time you are about to grab him, he pops up somewhere else.

NOTE: See *"COTS Lessons-Learned in the GSA Trail Boss Course"* in Addendum A to this chapter.

To alleviate this burden, traditional support approaches must be tailored to accommodate the COTS difference. Your approach should be commercially oriented to tap into the support infrastructure vendors establish to support their products. The 1991 report claims ***the goal is not to become locked into sole-source contractor support for COTS***, but to take advantage of the competition inherent within the commercial sector which keeps prices low and quality high. [COTS91] To survive in a competitive market, vendors are

intensely attuned to their customers' needs. Changes in your requirements will either show up in the next release of their software — or in that of their competitors.

NOTE: COTS products need not be in Ada, but they must be fully compatible with an open systems environment and must be accompanied by an assurance that they will be maintained by the COTS supplier for the life of the system.

Another concern with COTS is the chance for introducing a commercial virus into large, interconnected military software systems. Therefore, do not use software acquired from electronic bulletin boards, the public domain, or shareware sources, as they may contain hidden defects (and/or viruses) that can result in system failures, loss of critical data, or compromised security. An additional hazard with integrating different software packages (not originally designed to work together) is that sometimes you get unexpected responses (or ***side-effects***) under stressed operational conditions. Thus, for COTS to be effective, they must meet *de jour* or *de facto* interface standards. Your RFP must emphasize COTS compliance with controlled interfaces (e.g., those developed by the IEEE, ANSI, ISO, or NIST) that allow for evolutionary software exchanges. The benefit in using controlled interfaces is that software can be swapped out to improve reliability, gain performance, or better address changing requirements without impacting the entire system. Because hardware and software components must perform as integrated units, NIST has identified four key **MIS interface categories** which help in hardware/software partitioning:

- Application program interface (API),
- Human computer interface (HCI),
- Information storage and retrieval interface (ISRI), and
- Communications interface (CI). [FAA94]

To mitigate **interoperability risk**, you must evaluate each offeror's ability to judiciously select the proper standards for each interface category. NIST has developed an **Applications Portability Profile (APP)** model to ensure interoperability on a systems-wide basis, and to help make informed decisions on which interface standards to include. This model is useful for intelligently applying interface standards to the specific software system being procured.

The approach for achieving robust, easily upgradeable software varies widely among domains and should be approached differently. A rule of thumb for DoD software domains is:

- For a desktop environment, use 100% COTS;
- For MIS and some C2 systems, expect to base 80% of the system on COTS; and
- For weapons systems and real-time C4 and C3I systems, maximize the use of COTS, GOTS, and NDI to allow technology upgrades of computer platforms and software components (such as display drivers, operating systems, database management, and communications) as technologies evolve. [FAA94]

The remaining unique components must be controlled at the interface level to allow for future upgrades with minimum impact to the system. When procuring COTS, consider the following:

- Require that COTS deliverables be included in the contractor's **Logistics Support Analysis (LSA)**. The LSA *[discussed in Chapter 11, Post-deployment Software Support]* must address contract and organic support for COTS throughout the system life cycle;
- Ensure sufficient documentation is provided for COTS software for life cycle operation and support;

NOTE: COTS documentation need not adhere to requirements which may be levied on developmental software.

- Support COTS software at the vendor's current revision level, unless upgrades will adversely impact operational capability;
- Use competitive commercial practices to the maximum extent when supporting COTS software. For COTS software in systems with a life cycle greater than five years, consider recompetition of logistics support contracts; and
- Obtain locally purchased COTS software from your requirements contract if it is designated as a mandatory purchase item.

Cautions About Modifying COTS

COTS software consists of unmodified software applications (except as intended by the vendor). ***Commercial markets, independent contractors, and vendors control the design configuration and support (i.e., enhancements, modifications, and upgrades) of***

COTS software — not DoD. There is a basic reason why we do not want to engage in the modification of COTS. If you change even a small portion of a COTS product, when the next version comes out your software will no longer be compatible or upgradeable to it. You will be faced with patching old technology, while advances in state-of-the-art pass you by. Thus, ***do not purchase COTS software with the intention of modifying it!*** By so doing, you negate its benefits and create a very substantial source of program risk.

When making your COTS decision, remember that DoD has learned the hard way why modification of COTS software is not a sound decision. Rationalizations for modifying COTS have included: some, but not all, of the users requirements were met with the COTS package; or the COTS package did not comply with Public Law, DoD regulations, Service regulations, policies, procedures, or current systems interface requirements. Although these sound like good reasons to do some tweaking of a COTS package, the consequences of doing so have been costly and often regrettable.

WARNING! Modifying COTS software can be hazardous to the success of your software development and downright deadly for follow-on support.

The **Depot Maintenance Management Information System (DMMIS)** is an example of a program that made the *change-the-COTS* decision. The DMMIS is based on the CINCOM MRPII COTS package. The team's original intent was to utilize this package unmodified. But after further analysis, they determined that the CINCOM package would have to be customized to satisfy the user. The decision to proceed with the COTS customization was based on the following issues: (1) the COTS package did not meet the original users' requirements; (2) the differences between CINCOM's manufacturing-based software and the Air Force's re-manufacturing-based processes; and (3) the need to interface with legacy software systems. Although the DMMIS team believes they delivered a product superior to the original COTS package in less time than if developed from scratch, they learned some valuable lessons on why modifying proprietary software is not recommended. The team summarized the consequences of their COTS modification and lessons-learned as follows:

- They had to pay individual COTS charges to license each application running on each mainframe.

- They had to pay maintenance on the customized COTS packages.
- They had to pay an integrator to maintain their customized COTS packages.
- Difficulties surfaced when they wanted to take advantage of new releases of the original COTS. Now they no longer have the luxury of simply download a new version because their customized COTS has to be retrofitted to accommodate any new versions — costing down time and money.
- Customizing the COTS was time/money consuming because they had to understand what was needed, understand how and why the COTS was coded the way it was, and then figure out how to change it to meet their specialized needs. *[As you learned in Chapter 11, Post-Deployment Software Support, these same activities, are why our legacy software maintenance cost burden is so exorbitant.]*
- The combination of having to purchase the original COTS, having to employ an integrator on a FFP contract (in addition to having unstable requirements) led to more difficulties. The FFP contract forced the integrator to make short-term, quick fixes, the cheapest way possible. This, they know, will end up costing more in the long-run. It also required frequent open discussions between the developer and the user to reach an understanding — causing requirements creep.
- They learned that good requirements definition is crucial. This should be the first step before deciding on whether to modify COTS or develop software.

Recommendations the DMMIS team made for other programs considering a COTS modification include:

- Have the vendor make the modifications (not always possible).
- Instead of modifying the COTS package to match policies and procedures, the policies and procedures should be modified to match the COTS. This requires high-level management intervention and oversight.
- When software is available that meets the users' needs, COTS makes sense. When nothing is available, it may be necessary to develop software.
- If there is no other choice but to modify a COTS package:
  - The COTS package should meet most "*core*" requirements. Additional customization should be modularized, taking advantage of CASE tools (both in the COTS package and in the modifications), as much as possible. Make modifications ***outside*** the COTS package (i.e., put a shell around it) by modifying the COTS inputs and/or outputs, or by providing additional

manipulations or enhanced data analysis so that new versions of the COTS package can *plug-n-play* with minimal changes by the integrator.

- Use a **cost-plus-fixed-fee (CPFF)** or a **firm-fixed-price-incentive-fee (FFPIF)** contract so the contractor proposes the best long-term solution.

REMEMBER: You can surround COTS with interim functional layers that modify their inputs/outputs, but DO NOT MODIFY COTS!

Data Rights

As discussed above, DoD is not in the business of modifying COTS software. Therefore, data rights issues are only applicable to developmental software. Computer software **data rights** are of great importance to both the Government and the contractor. According to the FAR, the term *"data"* simply means *recorded information*, including software. *"Computer software"* means computer programs, computer data bases, and the documentation thereof. Regulations governing the rights to these data are found in FAR Sup 52-227-14, *Solicitation Provisions and Contract Clauses*.

NOTE: Purchasing COTS software should be your acquisition strategy only for those components that do not require change or maintenance by the Government. Thus, data rights for COTS will not be required.

Software is considered **technical data**. The allocation of rights to software and technical data represents an important issue governing the use of products developed or delivered under contract. The Government wants to ensure it has sufficient rights to enable it to use, maintain, and upgrade software and data. The contractor wants to ensure that the company's proprietary rights in software developed at its own expense are protected to maintain a competitive advantage. [MARCINIAK90] One advantage the Government seeks in acquiring data rights is the reduction of dependence on one contractor for life cycle support. Data rights are specified in three categories:

- **Unlimited data rights** allow the Government to use, disclose, reproduce, prepare derivative works, distribute copies to the public, perform publicly, display publicly in any manner for any purpose,

and to leave or permit others to do so. A subset of unlimited data rights are **government-purpose license rights,** with which COTS software developers are more comfortable. With government-purpose license rights the Government has the right to use, duplicate, or disclose data, including software. This usage is allowed if the contact is in the Small Innovative Research Program or established for government purposes only, either directly or through others. [MINUTILLO94]

- **Limited data rights** allow the Government to specify in the contract the delivery of limited rights to data that has been withheld or would otherwise be withholdable.
- **Restricted data rights,** particularly as they apply to software, limit what the Government may do with the software after the contractor has identified such data rights. Restricted rights usually are granted when the contractor has already developed all or part of the software at their expense, is declaring confidentiality, or owns the software copyrights.

Software submitted with **restricted rights** under government contract may not be used, reproduced, or disclosed by the Government unless it is:

- Copied for use with computers for which it was acquired,
- Copied for use in a backup computer if the computer for which it was acquired is inoperative,
- Copied for archiving, safekeeping, or backup,
- Modified, adapted, or combined with other software (however, the new, modified, or combined software is subject to the same restricted rights listed here), or
- Copied for use in (or transferred to) a replacement computer, including use at any government installation to which the computer may be transferred.

CAUTION! The entire issue of data rights is very esoteric. This discussion is intentionally general, as data rights are a sticky, controversial subject. It is recommended that you consult your contracting officer and/or legal advisor about your specific acquisition to flush out all data rights issues and alternatives. *The primary concern should be the capability to maintain the software during its government-use life cycle at a reasonable cost (i.e., data rights per se are not the issue).* The objective is life cycle supportability.

SPECIAL SOFTWARE SOURCE SELECTION CRITERIA

There are special criteria, uniquely software-specific, that if included as evaluation factors for source selection, will greatly increase your chances for selecting the **best-value contractor** for your acquisition. **Mosemann** reminds us that, "***We are buying process as much as product!***" He explains that ***a mature software development process is the most important factor in producing a quality product.*** [MOSEMANN94[2]] Without the process, the desired product cannot be produced. The process is critical to program success, and the process that matters is the one in use by the offeror. Although the June 1994 **Perry Memo** sets policy that **MilSpecs** and **MilStds** (especially process standards) are to be used sparingly *[if at all!]*, it does not mean process is any less important. Both your acquisition strategy and your RFP should specify that your source selection is based on a **best-value analysis** of all responsible offerors.

As itemized below under "*Selecting the Last Team Member*," RFPs for major software-intensive systems should include requirements for offerors to describe their software development process and capabilities. Offerors should identify a robust and mature Ada software engineering environment (SEE) and their applicable prior experience using that environment; an automated process control tool; relevant and pertinent domain experience; the extent to which peer inspections, Cleanroom engineering, and other ***best practices*** will be used; a Metrics Usage Plan; a specific Reuse Plan; and, if available, one or more proposed architectures in executable Ada code. Offerors should also be required to describe the risk management methods they propose to use; the development teams' roles and relationships; simulation, modeling, prototyping and demonstration plans; an approach to system and software design; training plans for software development personnel; defect prevention and quality management; an interface specifications plan; requirements traceability and testability plans; and safety and security considerations. Offerors' plans should also include: a schedule of objectives, accomplishments, entry and exit criteria; proposed contractor/government oversight reviews and the purpose of each; an earned-value performance evaluation plan; the extent to which the Government will have on-line access to their software engineering environment (SEE) and automated process control tool; and support documentation to be delivered.

NOTE: In your RFP, these requirements need not be more elaborate than the words in the above paragraph. The objective is not to tell the offeror how to do his job; but to communicate to them the information elements you need to equitably evaluate and compare their proposed process, methodology, tools, and other software development capabilities with those of other offerors.

As you learned in Chapter 1, *Software Acquisition Overview*, there are two additional key factors on which all the other evaluation criteria crucially depend and without which the other capabilities will not happen. They are ***key software development personnel*** and ***management commitment.***

Key Software Development Personnel

In talking about professional skill, Dwight D. Eisenhower, while General of the US Army, explained:

> *It is a long tough road we have to travel. The men that can do things are going to be sought out just as surely as the sun rises in the morning. Fake reputations, habits of glib and clever speech, and glittering surface performance are going to be discovered.* [EISENHOWER67]

Attention to **key personnel** on the organizational chart is another way to assess contractor capability. Be aware of the old *"bait-and-switch"* routine. Changing key personnel typically occurs after contract award for any number of legitimate reasons. It also occurs when an offeror cannot find the right people, or does not intend to use the ones proposed other than to win the contract. ***If key people are not current employees at the location where the work is to be performed, you may never see them.*** [DELLER94] Remember, key people are more important to the success of your program than just filling in the blanks on your matrix. One way to ensure personnel are obligated to your program is to require in your RFP that **Letters of Commitment** be provided from those individuals whose resumes are submitted with proposals. You should also require an explanation of any changes to predefined key positions in your contractor's periodic status reports. In addition, be cautious when evaluating a company with multiple divisions. ***Experience or success in one division does not necessarily represent the capability of another.*** Software engineering experience, plus domain experience, is key.

Skills Matrix

MITRE has developed a **Skills Matrix** methodology *[reference Software Management Guidelines, ESC-TR-88-001]* to help you determine the skills and educational requirements you need for your program and to relate those requirements to offerors' proposed personnel qualifications. The method is quite simple. First, knowledgeable government software engineers must develop a detailed list of the skills the job requires. Then, a set of threshold values of experience based on similar programs is established. Next, the offeror's list of resumes for those people they plan to assign to the program (submitted with their proposal) is graded against your skills (experience) requirements. Matrices showing each person and their experience (*yes* or *no*) can then be built. Finally, a summary against the threshold values can be prepared and evaluated. If in the course of the contract the developer reassigns people to or from your program, the matrix should be updated and glaring gaps in team experience identified.

Management Commitment

Remember, ***a quality software process is dependent on the quality of the people who implement it.*** There are never enough good software professionals, and even if you have them, there is a limit to what they can accomplish. If they are working overtime under chaotic conditions to meet unrealistic goals, they will be hard pressed to handle any greater challenges or to increase the quality of their products. Companies that implement process improvement techniques enhance the talents of their people. They enable their managers to understand where help is needed and to provide that support to their software teams when required. Process improvement programs enhance communication among professionals in concise, quantitative terms. They enable the flow of knowledge and minimize the time wasted on problems that have already been solved. They also provide a structure for software developers to understand their performance and ways to improve it. The company with a quality process has a smooth flowing, professional environment that **improves productivity** and avoids the enormous wasted efforts spent on fixing and patching team mistakes. [HUMPHREY90]

CHAPTER 13 Contracting for Success

WARNING! Offerors will tell you anything they think you want to hear. Structure your RFP so that you can ascertain whether their plans represent an on-going corporate initiative and commitment, or whether it is something they have concocted simply to satisfy the requirements of your RFP.

ATTENTION CONTRACTOR PERSONNEL! Why not encourage your top management to read these Guidelines?

SOURCE SELECTION

As discussed in Chapter 12, *Planning for Success*, ***"doing business like business"*** was introduced by **DSMC** as a solution for improving the DoD acquisition process. In line with this concept is a slogan written in large letters, visible to all travelers taxiing up to the main terminal at American Airlines corporate headquarters, Dallas-Fort Worth International Airport:

> *Cooperation, communication, mutual respect, and trust.*
> *Working together for excellence.*

This **team building** motto was adopted in 1987-88 during the era of deregulation, price cutting, and increased airline competition to instill the importance of **teamwork** among all employees. Valuing the ideas expressed in these words has helped empower American Airlines' employees to maintain their company's standing among US carriers. [MERRINER93] Selecting a development contractor you can respect, trust, and with whom you can communicate and cooperate, is a major milestone in achieving program success and software excellence.

NOTE: The GSA has published two reports: *Improving Industry/ Government Communications in Major Information Technology Acquisitions* and *Communications Between Government and Industry: A Reference Guide for Federal Information Processing (FIP) Resources Acquisitions*. See Volume 2, Appendix A for information on how to obtain them.

Consideration of these issues does not imply that contractor organizations are disrespectful, dishonest, hard to work with cutthroats, or that their employees are contemptuous, lazy, thieving, cheats. Cooperation, communication, respect, and trust are fragile commodities that must be established slowly and carefully and can evaporate quickly. The necessary conditions required for these high performing team attributes to occur include the following.

- **Contracts history**. The offeror has never done anything on any other contract to make you distrust them, or if they have, their most recent accomplishments are sufficiently successful to supplant their earlier shortcomings. They should furnish historical data (size/time/effort) on 3-4 completed programs in the same domain and of comparable size and complexity as your proposed program. These data should then be used to assess bidder productivity.
- **Understanding of the problem**. The offeror knows what you want them to do, and has convinced you they know how to do it. They should provide an estimate of software *quality* (expected defects remaining) and *reliability* [mean-time-to defect (MTTD)] at delivery, along with an estimate of the additional time required to make the software 99.9% free of latent defects.
- **Awareness**. You know their plan for meeting the requirement and are aware of their progress. Furthermore, they make sure you are aware of what they have accomplished, what the risks and problems are, how they are going to mitigate risks and solve problems, and where they are going. [HUMPHREY90]

The selection of a source(s), and subsequent award of a contract(s), follows a structured process which is designed to ensure impartiality while identifying the **best-value** for the Government. To solicit the best response from industry, offerors must know what discriminators you intend to use when they make their decision on whether to bid. The source selection process is illustrated in Figure 13-7. As with every other step in the software development process, ***planning is the key to contracting success***.

Source Selection Team Building

A key requirement for source selection is for the **Source Selection Evaluation Board (SSEB)** chairperson to assemble a qualified team of evaluators. The SSEB includes software experts in specialties such as software engineering, software architectural engineering, operating systems, compilers, software quality management and

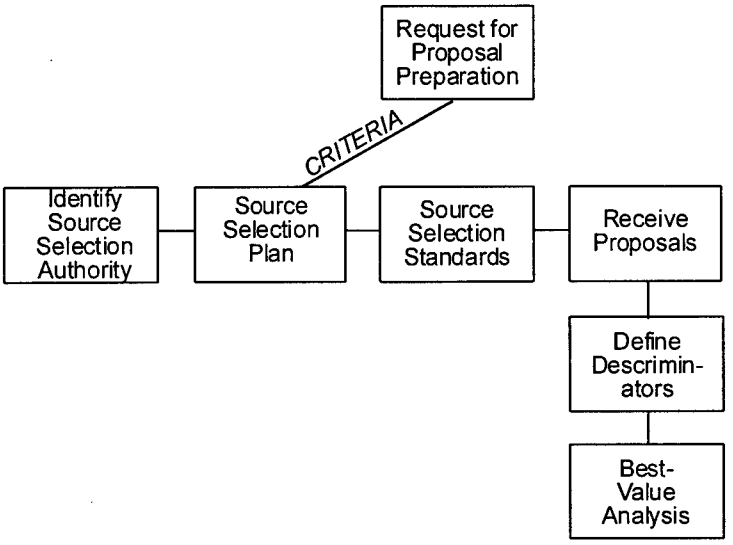


Figure 13-7 Source Selection Preparation Process

measurement, and database management systems and applications. SSEB membership should also include functional user representatives and the **Software Support Activity (SAA)** and other **Computer Resources Working Group (CRWG)** members.

Source Selection Planning

Source selection planning is normally worked as an integral part of the RFP preparation process. It provides policy and procedures for developing the source selection plan and proposal evaluation. Government software acquisition source selection should include a judgment on the validity of each bidder's proposal based on a comparison with the Government's baseline plan *[discussed in Chapter 12, Planning for Success]*, and the bidder's demonstrated performance (i.e., historical corporate software development success).

Pre-Validation Phase

In addition to an SEI capability maturity rating of a Level 3, the **Pre-Validation Phase** includes site visits and demonstrations as part of proposal evaluation (for NDI programs). The past performance of offerors involved in this phase is a key discriminator for participation in the next phase (e.g., BAFO). If the company cannot demo, they

should not be allowed to submit a proposal, or if allowed to submit a proposal, but subsequently fail the demo, they should be automatically eliminated from further consideration. You do need to ensure that the demo is cost conscious, i.e., do not make them overspend which will cause complaints from companies that they cannot afford to play. Of course, you are going to *pre-qualify*, and if there is a manageable number of offerors, the you might be in the position to *fund* (reimburse) the cost of the demo.

Selecting the Last Team Member

Selecting the right contractor is the most important decision you will make as a program manager. You can be the best program manager DoD has ever had, have the best strategies in place, and still fail if your contractor cannot meet program requirements.

Selection criteria and/or discriminators offered throughout this chapter can be grouped into technical, managerial, and cost categories. Some apply to more than one category. Source selection criteria you should consider when making this vital decision, based on offerors' responses, include (but are not limited to) the following:

Technical

- A well-written, comprehensive draft Software Development Plan (SDP),
- An approach towards management of an evolutionary requirements process,
- An understanding of CSCI control and management,
- An understanding of the technical/functional elements comprising the software they propose to build,
- A robust Ada development environment (**Rational Environment™** or equivalent) and applicable prior domain experience,
- Adoption of DoD-sanctioned SEEs and development tools, or equivalents,
- Automated process control software (**Process Weaver®** or equivalent),
- A plan to comply with DoD open systems standards,
- Proposed reuse of both program-developed and Government-approved software assets,
- A plan for training development team users, maintainers, and software professionals (as needed),
- A plan for risk identification, assessment and management,
- A plan to produce current source code documentation mirroring delivered code,

- A plan for clear, up-to-date user documentation,
- A plan for compliance with each CDRL,
- A plan to deliver data in DoD-approved format,
- A comprehensive SQPP,
- A plan for tailoring the use of MIL-STD-498 (if its use is proposed),
- An understanding of the interface and interrelationships between the software and hardware,
- Acceptance criteria for software deliverables that includes a criterion for program reliability expressed as MTTD at the time of delivery,
- Adequate software test resources and a plan for their application,
- A plan to incorporate COTS software and reuse to meet overall software needs,
- The proposal of a specific architecture(s) in executable Ada code using UNAS (or an equivalent middleware product),
- A plan to provide an Ada-driven demonstration as part of their proposal,
- A plan to constantly improve the software development process (SPIP), and
- A high score on the SCE or SDCE (a Level 3 or better).

Management

- A Software Development Plan (SDP),
- A current SEI Level 3 or SDCE equivalent rating,
- Relevant domain experience,
- A satisfactory Skills Matrix,
- Provision for on-line access to their development environment,
- Provision for on-line access to their process control tool,
- Description of peer inspection usage and extent,
- Corporate quality control processes and plans,
- Corporate management commitment to the proposed program,
- A plan to make supportability a software development priority,
- Plans and procedures for compliance with proposed reviews and audits,
- Metrics Usage Plan, purposes for which metrics will be used, and experience and knowledge with government-specified metrics and indicators,
- Software development scheduling tools and schedule realism, importance, and detail,
- Bonus points for a status room approach to software development and Government visibility (and/or on-line government access to the SEE),
- A comprehensive Subcontractor Management Plan, and
- An agreement to unlimited data rights for delivered software.

Cost

- A contract WBS reflecting the offeror's understanding of the requirement.

Obviously, technical and managerial criteria have significant impacts on cost. These must be folded into the appropriate selection categories to which they apply.

Navy Seawolf Lessons-Learned

The largest Ada software development in the US Navy, the AN/BSY-2 software for the SSN 21 Seawolf submarine, is estimated to cost $1.4 billion, is made up of 4.6 million SLOCs (of which 3.0 million will be new and 1.6 million reused and COTS), and is being built by over 600 software personnel. The AN/BSY-2 software component has come under GAO scrutiny "*as being exposed to unnecessarily high risk by rushing production to meet milestones.*" [JENKS92] Lessons-learned from a program of this size and complexity are invaluable for software RFP preparation. Recommendations from Seawolf lessons-learned include:

- Review subcontractor agreements against the prime contract for contractual consistency.
- Require early identification of all commercial products and associated licensing agreements.
- Mandate independence of the quality assurance organization from contractor development organizations.
- Require that the contractor establishes a system performance model upfront to be maintained throughout the program.
- Expand database design documentation requirements to include a logical, as well as, a relational and physical design.
- Clearly define firmware documentation deliverables.
- Require standardized software Style Guides for technical documentation produced across multiple developer sites. Make the prime contractor establish a single point-of-contact as the Style Guide distributor.
- Include electronic format as an optional contract delivery medium.
- Require that the contractor provides analyses of metrics data with respect to the SDP.
- Require installation of encrypted links between software developers and prime contractor sites.
- Require the prime contractor to establish controls for the management of common code and the document control process in software developers' SDPs and documentation.

- For older contracts still compliant with DoD-STD-2167A, requirements should be scaled down for software support deliverables and design documentation.
- Require the use of a standard set of support tools across development teams (e.g., compiler, code counter, document generator).
- Require strict version control of support tools throughout development to ensure all delivered software and firmware is compiled or assembled under the baselined version.
- Require the prime contractor to implement a shared problem reporting system across all developers early in the program.
- Evaluate hidden risks associated with the use of COTS tools (e.g., limited life cycle vendor support, upward incompatibilities, and schedule impacts associated with porting tools).
- Require the establishment of common databases to track all prime item development specification (PIDS) requirements for flow-down, and to ensure consistency across interfaces during integration.
- Require that configuration management tools are capable of supporting rapid turnaround during integration and testing.
- Make sure the contractor allocates additional time and resources for tool modification and for resolution of COTS interface/performance problems.
- Encourage the use of a standards checking tool to ensure high quality, maintainable code. [KING93]

Proposal Evaluation

The **source selection team** is responsible for the technical, managerial, and cost evaluation of each proposal. The technical evaluation must be rigorous, especially for system software, and alternative proposals must be evaluated for realism and value. It should thoroughly assess each offeror's software experience, Ada capabilities, tool availability and capabilities, and general software development and product assurance capabilities. It must also consider software supportability issues by fostering product-line sustainment where appropriate. Those offerors chosen for **BAFO** should be required to submit to a **software development capability evaluation** performed by the source selection team (and DoD personnel skilled in these evaluations). An SCE below Level 3 presents high risk and should be color-coded downward on technical (or both technical and managerial) evaluations.

Using the evaluation criteria, the SSEB must evaluate each offeror's understanding of RFP requirements, responsiveness to RFP requirements, and resources to perform as proposed. The SSEB

should also develop independent development schedule and cost estimates (based on modern software engineering principles), and compare this estimate with the contractor's estimates and proposal to identify areas and levels of risk.

CAUTION! To avoid too many nuisance changes to the contract and to preclude the contractor from using the inflexibility of "*contractually specified processes*" as an excuse for not meeting schedules, overrunning cost, or not meeting performance requirements, only the critical, top-level portions of the offeror's proposal should be made part of the contract.

Best-Value versus the Cost of Poor Quality

When it comes to source selection, **cost** has always been a major consideration. Too often, it has been ***the driving factor***. If you consider all the selection criteria and discriminators discussed herein, and believe you have identified the *best* developer who can deliver the highest quality product on a predictable schedule, the *"value of predictability and quality"* should influence your decision. ***Remember, the Government is as interested in buying a sound and predictable process as an attractively packaged and well-described product.***

Best-value. **Best-value** is an acquisition buzzword that grew out of the 1991 Defense Authorization Act. ***Best-value removes "cost" as the only criterion for source selection***. Instead, it gives the Government the ability to buy the *"best solution"* to its needs and to make smart business decisions. [POWER93] It gives industry the chance to be innovative in developing a solution and reduces the pressure to be the low-cost bidder, which usually results in taking too much risk and/or knowingly underbidding with the hope of *getting-well* after contract award. [WAYS94] It is especially applicable to software procurements where the lowest cost solution is not always the lowest risk.

Cost of poor quality. Hungry for business in a shrinking defense market, companies are often prone to *"buying-in"* on programs for which they have little ability to deliver. Be leery of those organizations who promise low costs, but do not meet even rudimentary SCE maturity levels or your other source selection discriminators. The ***cost of poor quality*** can be significant (in terms of scrap and rework

expense) when a contractor has to perform a process more than once to complete the work correctly. [ZELLS92] Even worse, you might have to write off $100 million or more when an unsuccessful program is terminated, as happened recently on several programs. The challenge is to quantify for source selection "*best-value*" purposes the surety that comes from selecting a contractor with a solid process, if he is not the lowest bidder. It can be accomplished. It is helpful to identify, in your RFP, the names of one or more consulting firms who will assist you in calculating this value. At the same time, you must be assured that the best-value proposal carries a reasonable price tag. Be aware, ***quality does not cost — it pays!***

Be aware, with best-value comes an added responsibility on the part of the Government to inform offerors in a clear, unambiguous RFP how you will evaluate them equally — in exactly the same precise manner. Otherwise, from the loser's perspective, you are simply choosing the contractor you want regardless of any other considerations. [WAYS94] You must clearly state whether you are buying the least cost, minimally compliant, or best-value. If best-value, you must state that technical solutions are more important than the costs associated with the program, while still working within an established budget. Remember, ***valid source selection decisions must be based on life cycle costs — not only upfront costs***. If your available funding profile is a constraint, it should be so identified to all offerors.

A NOTE OF WARNING about the final bidding and pricing process. To remain competitive, offerors often reduce quality assurance manhours to shave costs. It is imperative that the SSEB team tracks any changes at BAFO and change technical/management evaluations as required. While the quality assurance requirement remains part of the contract, performance in this area may be jeopardized. [MARCINIAK90]

PROTESTS

The **Information Technology Management Reform Act of 1995**, as contained in the FY96 Defense Authorization Act, has set in motion a process whereby the **General Services Board of Contract Appeals (GSBCA)** as a protest forum for information systems protests will be replaced by the GAO. The actual transition of jurisdiction is not expected to occur until August 8, 1996. The GSBCA plans to remain active in accepting protests until August 7th. Any acquisitions to which the Brooks Act *[see Chapter 2, DoD Software Acquisition Overview]* would have applied and which have been placed under the GSBCA prior to August 8th may have their adjudication completed by the GSBCA. There is speculation that, as the GAO assumes these new responsibilities, they may function somewhat more like the GSBCA. No substitute guidance has been issued that relieves DoD acquisition program offices from maintaining documentation suitable for protest proceedings discovery should such action occur.

To avoid protests, you must be careful when crafting your best-value RFP. You must articulate precisely what you mean by ***"value."*** Vendors need to know about your program, its mission, goals, and objectives. They must understand how this software purchase plays in the accomplishment of DoD's mission and in what user environment it will operate. You must state your needs in functional terms as much as possible. Without this background, offerors are likely to be in the dark about what you value most, and therefore, about how you will evaluate their proposals. [PETRILLO93] Clear, explicit detail must be included in your RFP about your definition of ***"value"*** and how proposals will be rated. The technical and cost relationships in award criteria and how they will be applied to the selection must be specified. **Cost** must be given weight along with **technical criteria** so there is not the impression of indiscreet flexibility (and thus subjectivity) in your evaluation.

NOTE: Also make provisions in your RFP, Section M, for assignment of value to product features or development approaches not anticipated.

Make sure your technical evaluators have ensured that every mandatory solicitation requirement has been met by the proposed awardee. A surprising number of protests are sustained because the awardee did not meet a mandatory requirement. Conversely, limit the number of

mandatory requirements in the solicitation to those that are absolutely necessary. A situation in which the solicitation calls for more than 500 mandatories is a recipe for protest.

Unfortunate experiences have shown that DoD cannot always rely on the integrity of even well-known vendors. **Best-value procurements** are particularly prone to error. In three recent major acquisitions, vendors have displayed the misconception that an agency can make whatever choice they wish in best-value procurements. This is only true to the extent that the decision is well-reasoned and documented. It is also essential that an agency follow its own rules. The GSBCA sustained protests where the solicitation states that an award would be based on a particular weighting of cost and technical factors, and where the agency did not follow that formula. *[See "Centel Systems versus Department of the Navy," GSBCA No. 12011-P, 1992 BPD, paragraph 359.]*

If sophisticated cost-technical tradeoffs are being made, it is essential that those making them have the necessary skills to perform the appropriate judgments and to do the quantifications required. This is especially true if the choice is the technically high-scored, high-cost solution. The case law does not require the source selection team to close the price gap between a high-tech, high-priced solution and a low-tech, low-priced solution. However, as a practical matter, it is very difficult to explain why you chose a $150 million *blue-blue* solution over a $100 million *green-green* solution unless you close the price gap in quantifiable terms. Sometimes this is accomplished through productivity studies and sometimes by evidence of other types of savings. There is no particular formula for quantifying a best-value price-gap closing, ***but it must be defensible.***

A mere list of desirable features along with the statement that these features are worth the extra money does not normally do the job. On one recent **Internal Revenue Service (IRS)** award, for example, the source selection decision lacked documentation. There was no detailed narrative statement explaining why the superior aspects of Company A's proposal were worth the large gap in price between its proposal and the other offers — from $500 to $700 million dollars. The only documented support for closing the price differential was something the IRS called "*Methods A & B.*" The agency's "*A & B*" approach attempted to quantify the differences between the relative power of the multi-user systems offered. The IRS attempted to express this difference in dollar terms, using what is often called a "*normalization process.*" Company A's mid-level systems were 6

times more powerful than Company B's systems. Because of this, the IRS analysts assumed that they would have to buy 6 times fewer systems from Company A as from Company B and discounted Company A's price accordingly. This simplistic analysis was rejected by the GSBCA because the IRS did not take into account the benefits the Government would receive from the different offers. The offers were made on the basis of the IRS' estimated quantities, and Company B did not offer 6 times as many systems as Company A. The second time around the IRS did a much more sophisticated job of measuring the comparative benefits of the different offers and the GSBCA, and ultimately the Federal Circuit Court, sustained their award.

Another recent case illustrates the problems of using a crude *normalization process*. It also illustrates the problems in relying on one approach in general. It is better to use several methods and to look at each one of them carefully. Often a price-gap closing will require a source selection team with a combination of information resource management and accounting skills of a high order. Institutional or personal arrogance in recognizing this requirement is often punished later on. The theory that "*we do not need help to do our own procurements*" is fine until the protest is decided adversely for DoD. Then no excuses are accepted. Do not assume that your in-house people are as knowledgeable as the extremely expensive experts who will be hired by disappointed vendors.

Be prepared to *"fight fire with fire,"* and do not be hesitant to bring on your own big guns. ***It is suggested that serious consideration be given to hiring an outside expert to "red team" the award decision before it is signed.*** This expert may wind up testifying for the Government. His or her insights will be extremely helpful in anticipating future challenges. SAF/GCP will help you secure such an expert if desired. In anticipation of the need to employ experts, you must identify your intentions in your RFP, including the identity of the experts, or firm of experts, you intend to employ.

Industry involvement. Keeping industry involved throughout the acquisition planning phases is also essential to a successful procurement. Up until the RFP is released, you should maintain an open, public dialogue with industry through industry briefings, by providing open reading libraries, allowing industry to brief you on potential solutions, and by releasing draft RFPs. One or more draft RFPs ensure that requirements are better defined and understood. Industry can better respond to draft RFPs than amendments to RFPs that cost time and money, or even worse, **multiple BAFOs.**

[WAYS94] You can require that offerors submit a cost and technical tradeoff analysis based on their understanding of your system's mission. You can also have them submit alternative proposals offering different solutions with low-cost/low-performance and high-cost/high-performance alternatives. [BRENDLER93] This will enable the **SSEB** to consider the implementability, applicability, and validity of each offeror's proposal.

Can an exponentially lower price be credible? Perhaps. In one recent acquisition a small, innovative company using **Cleanroom engineering** *[see Chapter 15, Managing Process Improvement]* bid $20 million. The next lowest bids were in the $120 million range. The lowest cost bid, which the source selection authority acknowledged had the best technical proposal, was not accepted because the cost was considered too low (and risky). In this instance, it may have been in the Government's best interest to make two awards — one for the $20 million proposal and another for a more conventional proposal. This would have served the dual purpose of reducing risk, and, if the Cleanroom solution had resulted in a successful product, demonstrating that revolutionary improvement in software productivity, quality, and cost is feasible.

CONTRACT AWARD

After completing technical, managerial, and cost evaluations, the program office sometimes conducts oral discussions with potential contractors to clarify any ambiguities in their proposals. If the offeror's approach is acceptable, it must then be made part of the contract.

Since peak manpower and staffing profiles so profoundly affect software development schedule, quality, and reliability, the ***lower limits*** on these two variables should be negotiated into all contracts. After offerors submit BAFOs, the winning contractor(s) is selected. Figure 13-8 (below) illustrates the contract award process. Once selected and awarded, the contractor becomes an **official team member** within your program management structure. To the winning contractor, after having scaled all the hurdles to contract award outlined in this chapter and that you have packed into your RFP, a famous quote by Vince Lombardi while coach of the Green Bay Packers says it all:

> *Winning isn't everything; it's the only thing!*
> [LOMBARDI68]

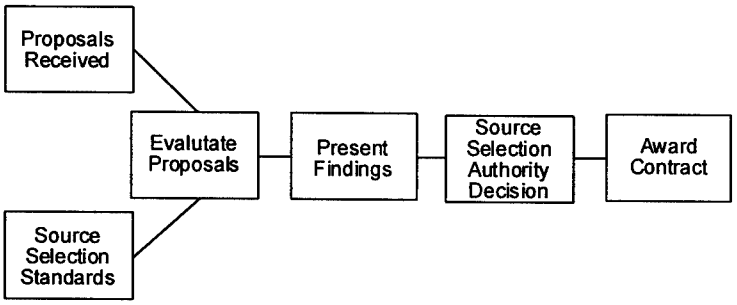


Figure 13-8 Contract Award Process

NOTE: Source selection standards must be approved by the SAA before RFP release.

Admiral Ernest J. King, Commander-in-Chief of the US Fleet, Chief of Naval Operations, and principle advisor to President Roosevelt during World War II, summed up what teamwork means.

> *Discipline is willing obedience to attain the greatest good by the greatest number. It means [the] laying aside, for the time being, of ordinary everyday go-as-you-please and do-as-you-like. It means one for all and all for one—* ***teamwork****! It means a machine—not of inert metal, but one of living men—an integrated human machine in which each does his part and contributes his full share.* [KING52]

Software success is achievable when government and industry walk in lockstep toward the team goal of a quality software development using the contracting mechanism. The type of contractual vehicle you choose depends on how much government/industry team interaction you require, and how integrated your human machine must be.

ATTENTION MANAGERS! Success requires a contractor with top-down commitment to software engineering practices, concepts, technology, training, planning, people, and the willingness to commit corporate resources to achieve quality goals. The best-value contractor you select must be able to provide you with a superior technical solution at a reasonable cost. You have been provided with the mother of all

software source selection checklists both in this chapter and in Volume 2, Appendix M. Use them and GOOD LUCK!

REFERENCES

[ALIC92] Alic, John A., et al., Beyond Spinoff: Military and Commercial Technologies in a Changing World, Harvard Business School Press, Boston, Massachusetts, 1992

[BAKER92] Baker, Emanuel R., "TQM in Mission Critical Software Development," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[BRENDLER93] Brendler, Beau, "Best-Value Unclear: Procurement Philosophy Explored at Meeting," *Washington Technology*, October 7, 1993

[COTS91] *Joint Command Commercial-off-the-Shelf (COTS) Supportability Working Group (CSWG) Final Report*, June 1991

[DELLER94] Deller, Bob, "Agencies Need to Be Wise When They Buy Expertise on Contract," *Government Computing News*, March 7, 1994

[DuPICQ80] du Picq, Col Charles Ardnant, Battle Studies, 1880

[EISENHOWER67] Eisenhower, GEN Dwight D., At Ease: Stories I Tell My Friends, Doubleday & Co., New York, 1967

[FAA94] "Report of the 'Open System Development' Subcommittee," briefing prepared by the Federal Aviation Administration, Research, Engineering, and Development Advisory Committee, Headquarters, Washington, D.C., 1994

[HERMANN94] Hermann, Robert, "Defense Science Board Task Force on Acquisition Reform," *Army Research, Development, and Acquisition Bulletin*, January-February 1994

[HILLIER95] Hillier, Pat, "Standards-based Development: Using Ada to Generate COTS Products," paper presented to the Seventh Software Technology Conference, Salt Lake City, Utah, April 1995

[HUMPHREY90] Humphrey, Watts S., Managing the Software Process, Addison-Wesley, Reading, Massachusetts, 1990

[HUNTER92] Hunter, Mark T, "Award Fee in Software Acquisition" (AFIT thesis: AFIT/GLM/LSY/92S-42), Air Force Institute of Technology, Dayton, Ohio, 1992

[JENKS92] Jenks, Andrew, "DoD to Change Purchasing," *Washington Technology*, August 27, 1992

[KELLEY93] Kelley, LTC Kevin C., "Marine Corps Is 'Proud to be First' During Prototype: Prototype Site Activation Occurred Two Days Ahead of Schedule," *J-CALS Connection*, Vol. 3, No. 3, PM J-CALS, Third Quarter 1993

[KING52] King, ADM Ernest J. and Walter Muir Whitehill, Fleet Admiral King: A Naval Record, W.W. Norton & Co., Inc., New York, 1952

[KRATZ84] Kratz, L. A., Drinnon, J. W., and Hiller, J. R., Establishing Competitive Production Sources: A Handbook for Program Managers, Defense Systems Management College, Fort Belvoir, Virginia, August 1984

[LAKE94] Lake, Jerome G., "Do We Need the Statement of Work?" *Program Manager*, Defense Systems Management College, Fort Belvoir, Virginia, September-October, 1994

[LOMBARDI68] Lombardi, Vince, quoted by Jerry Kramer, *Instant Replay*, 1968

[MERRINER93] Merriner, Captain Bill W., information provided by American Airlines Boeing 767 captain, Corrolitos, California, May 10, 1993

[MICOM91] *Methodology for the Management of Software Acquisition*, US Army Missile Command, Software Engineering Directorate, Redstone Arsenal, Alabama, June 1991

[MILLS95] Mills, Andy, "Software Acquisition Improvement: Streamlining Plus Risk Management," paper presented to the Seventh Annual Software Technology Conference, Salt Lake City, Utah, April 1995

[MINUTILLO94] Minutillo, Daniel C., "Understanding of Contract Terms, Policies Helps Repel Licensing Mistreatment," *Federal Computer Week*, January 10, 1994

[MOORE] Moore, Richard, "A Software Management Perspective," briefing presented to BOLDSTROKE

[MORROCCO94] Morrocco, John D., "Arms Modernization Key Long-term Goal," *Aviation Week & Space Technology*, March 14, 1994

[MOSEMANN93] Mosemann, Lloyd K., II, as quoted by Joyce Endoso, "AF Official Says Buying Reforms Will Come: COTS Use, Higher Thresholds Included in Plan for Changing Regulations," *Government Computer News*, September 20, 1993

[MOSEMANN94[1]] Mosemann, Lloyd K., II, comments provided to AFPAM 63-115, May 1994

[MOSEMANN94[2]] Mosemann, Lloyd K., II, "New Frontiers in the Acquisition of Software," briefing presented to the Defense Science Board, 1994

[PATTON47] Patton, GEN George S., Jr., War as I Knew It, Houghton Mifflon Co., 1947

[PETRILLO93] Petrillo, Joseph J., "If Agencies Won't Articulate Needs, Can There Be Best-Value?" *Government Computer News*, October 25, 1993

[POWER93] Power, Kevin, "Dark Science? Course Takes Mystery Out of Best-Value Buys," *Government Computing News*, December 6, 1993

[RAND95] Rand, Linda M., "ReARC COTS Software Integration Lessons-learned," briefing presented on July 24, 1995

[SHALIKASHVILLI94] Shalikashvilli, GEN John, as quoted by John D.

Morrocco, "Arms Modernization Key Long-term Goal," *Aviation Week & Space Technology*, March 14, 1994

[SPAT92] "Software Process Action Team, Process Improvement for Systems/ Software Acquisition," Air Force Systems Command, Final Report, June 20, 1992

[SULLIVAN94] Sullivan, Bruce E., "The Section 800 Report: Streamlining Defense Acquisition Law," *Army Research, Development, and Acquisition Bulletin*, January-February 1994

[VELOCCI91] Velocci, Anthony L., Jr., "ATF Development Program's Risk Light for Lockheed, Financial Officer Says," *Aviation Week & Space Technology*, May 13, 1991

[WAYS94] Ways, John P., "Best-Value Not the Best Approach: Procurement Strategy is Well-intentioned But Hard to Implement," *Washington Technology*, February 10, 1994

[ZELLS92] Zells, Lois, "Learning from the Japanese TQM Applications to Software Engineering," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

Blank page.

CHAPTER 13
Addendum A

Lessons-Learned in the GSA Trailboss Course

George Coulbourn
Boeing
Barry Ingram
EDS

BACKGROUND

In 1988, the GSA introduced a two week course known as Trail Boss. The program was designed to train Government personnel in the conduct of information technology (IT) acquisitions. The original goal was to train approximately 300 people. After 10 classes this goal was met; however the demand continued, and in 1995, with the advent of class number 20, over 600 people have received the training. Beginning with class number 3, industry participation was requested through the auspices of ITAA, the Information Technology Association of America. The authors were privileged to conduct most of the industry sessions during this period. Many other members of ITAA participated in various presentations over this time period. The materials prepared for that purpose have evolved and have been used in numerous presentations by members of ITAA and by others. Most presentations have occurred in an interactive setting, such that problems and impediments could be openly discussed and analyzed. As a result of these experiences, Government and industry have learned many lessons. The purpose of this paper is to capture the more pertinent lessons in hopes that by their promulgation, both Government and Industry will benefit.

Industry Trail Boss Presentation Approach

Fundamentally, the ITAA presentation is about mutual understanding and human and corporate behavior. We discuss processes and problems from the perspective of an independent systems integrator. We present a detailed description of how we make our procurement bid decisions. We describe the acquisition process from our perspective and discuss our objectives and concerns at each stage. In addition, the perspective of the subcontractor is presented and actual cases are discussed to provide real examples. We encourage and usually obtain interaction with the audience. Depending upon the audience and the time available, various topical issues may be addressed. Finally, lessons-learned by us are presented in the form of recommendations for consideration by acquisition teams, and the input of the participants is taken by the industry presenters for a better understanding of the Government's issues.

LESSONS-LEARNED

Following is a list of lessons-learned by the authors. They are a result of more than one hundred presentations, made together and separately, on the subject of IT acquisitions over the past five years. An appropriate disclaimer regarding the selection criteria, completeness, and presentation is hereby made: these lessons are presented in a format that could be presented as recommendations to a Government team delegated the responsibility of acquiring a large IT system. Some observations are controversial. All pass the authors' tests of being legal, achievable (albeit difficult) and mutually beneficial to Government and industry. No attempt is made to prioritize them.

1) Obtain top management support before proceeding

IT system acquisitions are difficult endeavors, at best, and impossible at worst. The process must conform to a host of laws, regulations, and policies that govern procurements in general, and then conform as well to another set devised exclusively for IT procurements. Occasionally, pressures from within the agency, from others in Government, or from industry can present obstacles that the acquisition team cannot overcome. Experience has shown that on almost all large acquisitions, there are times when success requires a tough decision by a senior executive.

The larger the procurement, the more players involved, and the longer the duration, the greater the potential for problems requiring executive action. When programs within agencies compete for funding and other resources, or when challenges to scope or other requirements arise, executive involvement is sometimes necessary. Furthermore, in a large, complex procurement, there simply are times when the authority to direct, countermand, or waive certain actions is essential to success.

Executive support should be obtained upfront. Obtaining executive participation and "*ownership*" should be an integral part of the acquisition strategy. The acquisition team should find an executive sponsor (or sponsors) and periodically review their acquisition strategy, milestones and risks. Care should be taken to highlight the major threats to success. The range of responses that might be required should be discussed to ensure that executive support accepts the exposure. Properly done, senior management is informed and ready to act when required. Finally, the Trail Boss program can help. Obtaining a "*Trail Boss*" designation from GSA requires a higher level of agency executive involvement than might otherwise be customary.

2) Consider a Congressional support strategy

Good programs can die without Congressional support. The authors have seen this occur many times in the past with reduced Federal budgets and close scrutiny of all programs. It is important to maintain high visibility of programs to ensure continued life. This support must be consistent and must last throughout the acquisition and program phases. It is important, therefore, that a good Congressional support strategy be developed and maintained. This may take the form of frequent briefings of schedule, funding issues, program threats, technology requirements, and mission objectives to Congressional staff. Reviews of the potential savings and advantages of the program can be given to highlight the program's importance. Executive level support from Agency management is vital in sustaining Congressional support. In fact, it's their job.

3) Involve your end users meaningfully and continually

The need to ensure end user involvement is so obvious that it might not warrant discussion except that, obvious or not, some acquisition teams fail to obtain it. Program success demands that the system be

accepted by end-users and that, by their use, the system performance objectives are achieved.

It is not particularly difficult, in principal, to obtain end-user involvement. The most difficult steps are the first ones: identification of a representative sample of the end-user community and obtaining their commitment to support the acquisition. If these two steps are done properly, the probability of success is enhanced considerably. If not, the risk that the system may not perform as expected, or not be accepted by the end-users may be high. Roles for the end-user representatives include the following:

- Help define the system requirements,
- Assist in prioritizing requirements,
- Assist in defining "*mandatory*" and "*desired*" features,
- Ensure that the requirements are captured in the text of the RFP,
- Help mediate conflicting requirements within the user community,
- Continually validate their decisions within the user community,
- Help determine whether to incorporate changes in mission, policy or technology into the process,
- Participate in risk assessment and mitigation decisions,
- Concur with any changes made either to requirements or policy during the process,
- Provide the end-user perspective during interfaces with the bidders, especially during any demonstrations, and
- Help prepare the user community for the changes that the system will bring.

To perform these functions, end-user representatives should serve on the acquisition team and play a meaningful role in the evaluation and selection process. The challenge for the acquisition team is to ensure that the end-user representatives remain a representative sample of the end-user community throughout the process.

4) Market to your vendors pre-RFP

Vendors need to be brought into the acquisition process as soon as a need is established and while the requirements are being developed. By getting industry involved prior to issuance of the RFP, they can offer technological and business advice without jeopardizing the procurement, since this is prior to any formal documents being formulated and communications being restricted. In addition, new technologies and capabilities not previously known or understood can be considered as possible alternatives.

While the contracting community is competing against each other for your business, Government, in turn, is "*competing*" for the attention of qualified bidders. Since contractors' resources and bid and proposal funds are limited, enticing qualified bidders to consider the program is critical to the successful accomplishment of the acquisition. This time period provides a unique opportunity for both industry and Government to look at possible alternatives and solutions in an open, noncontentious environment.

5) Develop a plan to use the RFC or DRFP effectively

The objective of the Request for Comments (RFC) or Draft RFP (DRFP) process is to gather information to prepare an RFP which best reflects the real requirements and fulfills the needs of the end-user, and to prepare the vendor community for the coming competition. Therefore, development of a plan to utilize these vehicles most effectively is essential. Some of the specific goals of the plan should be the following:

- Improve the overall requirements definition,
- Include all anticipated sections of the RFP for a more complete review,
- Minimize questions and surprises after the RFP is issued,
- Minimize ambiguities in the RFP,
- Minimize delays and changes,
- Get recommendations on improving the RFP, and
- Attract qualified bidders.

Changes and improvements in the solicitation made at this early stage of the procurement process contribute to a much smoother process later on. Just as in a software development program, time spent on the front end of the effort to completely define and document the requirements and scope of the program results in lower overall costs and time expenditures. Conversely, the cost and time required to revise the designs and requirements after the RFP release are very high, both to Government and Industry. Changes later in the program may require bidders to adjust teaming arrangements, re-engineer solution designs, and even reverse previously positive bid decisions.

6) Use experienced qualifiers

It is to the Government's advantage to get only qualified bidders. This is especially important on large contracts with high mission risk. Therefore, developing and requiring certain levels of experience or

proven capabilities is a valid means for qualifying prospective contractors. This may take several forms:

- Past team experience on contracts or programs of similar scope and magnitude,
- Proven team capability in a particular technology,
- Proven software development capability,
- Documented software engineering maturity,
- Corporate size to absorb and compensate for risk inherent in the program,
- Adequate numbers of qualified staff with pertinent experience on the team,
- Proven program management experience,
- Capability to provide global support, and
- Capability to run a Live Test and Demonstration.

While these may be seen by some as limiting to competition, they are important criteria in the selection of any qualified team to ensure success. They do not prohibit smaller contractors from bidding, since they have the opportunity to become players on larger teams. In fact, this teaming may provide them access into some new areas. Furthermore, by clearly stating the qualifications expected of bidders, vendors can better gauge the appropriateness of preparing a bid and subsequent protests may be avoided. However, the Government should also be sure that any "*qualifying*" requirements are actually needed and provide a real advantage to the program. We have found that non-value-added requirements may eventually get removed from the final list of mandatory requirements and that they may have only added cost without benefit.

7) Use functional (performance) specifications

Historically, many sets of procurement specifications have been "*prescriptive*," meaning that RFPs ask for specific products or products with specific hardware capabilities: typically a commodity type of product, for example a video display device with a 1024 X 768 pixel resolution. Another form of the prescriptive requirement is to prescribe exactly how you want a service performed, rather than the end result of the service. While this may be the preferred method to acquire commodity products, it is very constraining when the procurement is for large or complex systems. Also, when the procurement duration is lengthy, prescribed products may become outmoded or may be overtaken technically by superior offerings. In

these cases, vendors may be unnecessarily limited in selecting products or attributes for their solutions.

The Government is usually best served by providing "*functional*" specifications which describe the requirement or need, and ask for a solution from industry, without specifying the actual methods, products, or technologies to be utilized. With this freedom, industry has the opportunity to offer new ideas and is not constrained by a particular technology. We realize that asking for functional requirements may make the evaluation more difficult, but the reward of a better, more current, or more effective solution may justify the increased effort.

8) Challenge complexity and non-value-added reviews and approvals

Most enterprises knowingly and unknowingly support non-valued-added activities. In industry today, considerable effort is being expended in eliminating work that does not add value as "*perceived by the customer*." Internal administrative work is important when it enhances the quality of the product. It keeps decision makers and supporters informed, but most such work does not meet the value-added test. Worse, it adds expense and diverts talent from other tasks, and worst of all, it increases the duration of the process.

Reducing non-value-added work requires that the acquisition team challenge the culture of their organization. The challenges are to written and unwritten policy. Experience has shown that the challenges can usually best be made at the start of the acquisition process, when the team is laying out its schedule, milestones, and developing its rules.

An approach that has met with some success is to define the acquisition approach such that reviews and approvals are combined, conducted in process, or eliminated altogether. The ability to do this is considerably enhanced if senior executive support has been obtained, and is facilitated if the executive has '*bough- in*" to the concept.

9) Distribute risk equitably

Contractors devote substantial time to risk analysis regardless of whether the RFP explicitly calls for such analyses. The reason is that all risk that must be borne by the contractor must be identified and planned for to ensure that an realistic business case is developed.

Contractors must either price or mitigate all elements of risk. The RFP is the basis for the risk analysis since it furnishes the bidder with the Government's apportionment of risk. During the acquisition phase, risk analysis and pricing by bidders is a business decision to which the Government is a party.

Every risk that the Government lays off on the contractor (that the contractor accepts) will increase the price from a responsible bidder. Hence, the Government itself should analyze each element of risk to determine, first, whether it can be managed by the Government; and then, whether it is more cost effective for the Government to assume the risk or pay for the contractor to do so. Even in instances where the Government cannot manage an element of risk, it may be less costly for the Government to assume it anyway if uncertainty will cause the bidder to assign a high cost to the risk element. Indeed, in the extreme case where neither the Government nor the contractor can manage an element of risk, the Government should always assume it; otherwise, the Government pays a premium for a service that cannot be performed and risks contract disputes. A lesson learned and relearned is that attempts by buyers in all areas of society to lay off unreasonable risk on their contractors backfire. Invariably, when difficulties occur, cooperation erodes, the contractor looks for a means of escape, the mission suffers and both sides lose.

Time spent by the Government, especially during the draft RFP stage, in understanding the potential bidders' perspectives concerning risk may be the most useful effort of all. Only by talking with potential bidders can the Government expect to understand how the bidders assess the risk. And, it is the bidder's assessment that matters at this point.

10) Really work hard on internal and external RFP integration

Internal integration means assembling an RFP in which the Sections (especially Sections C, H, L, M and the Technical Specifications) are consistent with each other. This is difficult to do because of policies, and sometimes law and regulation, that mandate inclusion of contract clauses unrelated to user and system performance requirements. It is also difficult when agencies prepare prescriptive specifications (i.e. specifications that prescribe elements or products, rather than performance requirements).

It is difficult to overemphasize the importance of this point. Ambiguity is the acquisition team's enemy. Ambiguity allows different

interpretations by bidders, and bidders are generally entitled to the minimum interpretation. Ambiguity can make performance evaluations and tests difficult or impossible. It can precipitate delay when amendments are required for correction.

Because the various RFP sections are typically assembled by different teams, integration among the sections is a separate activity that must be planned *before* the sections are prepared, and completed *afterwards*. The planning starts with the recognition of the need and the complexity of the task. Care must be given to the preparation of guidelines and standards for RFP section writers. Interim reviews are important. Independent reviews can also help. Electronic tools for RFP decomposition or "*shred*" (employed routinely) by bidders can assist in expediting the task of comparing families of requirements. All of the above takes time and time should be allowed in the schedule.

External integration means assembling an RFP that is consistent with the overall acquisition strategy, the mission of the system, the mission of the agency and realities in the agency's internal and external environments. It is also concerned with establishing the kind of relationship desired between the agency and their contractor. The apportionment of risk between agency and contractor is critical to the contractor's behavior. Hence the contract type, evaluation criteria and methodology, mechanisms for change and technology refreshment, performance measures, and penalties, if any, must be dealt with consistently.

11) Publish detailed evaluation criteria and methodology

There are two powerful reasons for publishing full and complete evaluation criteria and for doing so as early as possible (draft RFP stage): first, to enable bidders to know what is really important to the buyer; and second, to enable the agency to avoid protests.

Why would an agency not want its bidders to know how it will decide from among its offers? Detailed evaluation criteria allow bidders to design their solutions in accordance with what really matters. Denying bidders the detailed information forces them to guess (*and they will guess!*). Accordingly, the process may favor bidders who guess more accurately, although they may have no more real knowledge of the importance of the criteria. Likewise, clearly stating the relative importance of detailed evaluation criteria will also ensure that the most important requirements are adequately addressed in the solutions offered.

Perhaps, the most effective approach to avoiding protests is to furnish detailed evaluation criteria, follow the criteria meticulously, and debrief bidders in detail in accordance with the published and practiced criteria. When bidders understand why they lost (and hence, why the successful bidder won), and believe that the decision was made fairly and in accordance with the evaluation criteria, the principal reasons for protest are neutralized. Published, detailed evaluation criteria allow the agency to accomplish this. However, it is necessary that the agency plan for this process from the start. In other words, the agency must recognize that the evaluation process has two products: the selection and the debriefings of the unsuccessful bidders.

12) Ensure that the evaluation methodology and criteria both ALLOW and REQUIRE you to select the best vendor

Obviously, the evaluation criteria, along with the technical specifications in the RFP, drive the solutions proposed. Therefore, it is incumbent upon the Government to evaluate the proposal in accordance with the methodologies and criteria stated in the RFP. Any deviation from these procedures, without good explanation and reason, will invite questions and invariably, protests. This may suggest that adequate thought was not put into the evaluation process, or worse yet, favoritism is being shown to another bidder.

It is just as important that the evaluation methodologies and criteria enable the selection of the proposal and bidder that best meets the Government's requirements for the benefit of the program. Any criteria that force a selection other than this need to be removed. Along this line, it is important that only the necessary requirements are specifically stated in the RFP. Extraneous requirements or standards that provide no real added-value to the procurement or add unnecessary complexity should be avoided.

Detailed evaluation criteria are also a valuable defensive tool for the acquisition team in maintaining stability in the event circumstances change. It is not unusual in a lengthy acquisition, for missions, technologies, or people to change such that pressure mounts to change the acquisition strategy without changing the RFP. In such instances, the evaluation criteria can become a welcome constraint for the evaluation team, requiring them to stay the course.

13) Tell your bidders everything

It is axiomatic that the more that bidders know about a customer's requirements, selection criteria and the environment in which the system will operate, the more closely they can design a system to met those requirements. Agencies penalize themselves when they withhold information that might materially affect design decisions made by a bidder. Bidders need to know not only the agency's best estimates of performance requirements, but also how those requirements relate to each order to be able to conduct meaningful trade studies during the design process. Trade studies involve not only technical designs, but management system operations and cost as well.

If information is not available, bidders will develop their own estimates. This can significantly increase the risk of wasted effort on the part of both bidders and the agency and increase the probability for protest. Most information that is denied bidders results from internal agency policy rather than law or regulation. For example, there is legal prohibition against providing bidders with cost data so that they can "*design-to-cost*," a common commercial practice. When this is done, most bidders will try to maximize their offering within the anticipated available funding, or bid somewhat less than the available funds to provide an attractive price. The advantages of "*designing-to-cost*" are many; solutions that meet yearly budget allowances with implementation plans in step with Government needs; more bidders within the "*competitive range*" from a cost standpoint; and, in many instances, innovative proposals offered which meet the requirements at substantially reduced costs.

Many mechanisms are available for information exchange, even after RFP release. Agencies need only to be careful that no bidder receives information not made available to all. In fact, the more information that is released to the entire bidding community, the better the quality and quantity of solutions that can be offered and the closer they should respond to the real needs. In addition, as more information is made available, there is less opportunity for unscrupulous individuals to attempt to provide or gain undeserved advantage. Prior to RFP release, and especially in the early stages of planning for an acquisition, agency and prospective bidders benefit from open exchange of information.

14) Don't drop the "curtain" until RFP release

IT system acquisition is a lengthy and complicated process; in many instances, unnecessarily so. However, until fundamental changes are made, bidders and agencies will have to live with it. The key is not to make the system any more complicated or difficult than it already is.

There may or may not be a single root cause for the difficulty and complexity of IT system acquisitions, but one thing is clear: the very long duration of the process exacerbates all potential problems. It is the long duration that allows product cycles to render agency requirements definition and bidder solutions obsolete. The long duration provides time for agency needs, missions, and environments to change, thereby invalidating requirements. Over time, people and policy change. Yet communication is cut off, sometimes for more than a year before the agency selection process is completed.

Agencies that make their people available for information exchange with prospective bidders as long as legally allowed will benefit most. Unfortunately, some agencies cut off communication well in advance of RFP release; some before release of their draft RFP. This penalizes both bidders and the agency. And, it is unnecessary.

15) Communicate with your vendors frequently after the "curtain" drops

After the RFP is released, and sometimes even after the RFC is issued, the Government severely restricts communications with the vendors, and then usually only in written form and through the contracting officer. This is understood, although not appreciated, by the bidders as a way to prevent inappropriate discussions and prevent advantages to some bidders. This pattern of communication is reflected in Figure 13-9 and compared with the communications in the commercial environment.

Notice how, in the commercial arena, communication actually increases as the program progresses. However, it is also important to keep communications open in the Federal procurement process with the entire bidding community after the "*curtain*" drops. It is essential that bidders feel that the program is moving, is under control, and is being pursued with as much enthusiasm as possible by both the program shop and the contracting office. One of the best indicators

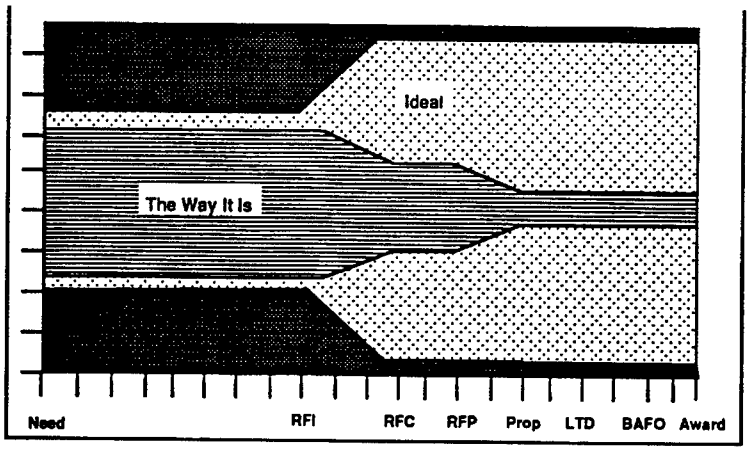


Figure 13-9 Level of Communications — Federal versus Commercial

of progress is frequent communications about the status of the program, questions and answers, and accurate milestones and event dates.

As the bidding community reviews the programs they are bidding, those with poor communications may be dropped, simply because the contractors feel they are not as critical to the Government as they are to the bidders. Internal struggles within bidders' organizations for continuing funding to bid programs are common. As programs drag on for months and even years, good communications of accurate information about status is critical to preserving both the prime and subcontractor teams.

Finding ways of maintaining communications serves the Government in other ways. For example, in any given acquisition there may be bidders who are incumbents or are otherwise serving the customer through other contract vehicles. If other bidders do not have a presence in the customer's business, they may be (and certainly believe themselves to be) at a continuing, sometimes growing, disadvantage regarding access to changes in the customer's missions, problems, and preferences.

16) Use LTDs only to reduce unacceptable risks and uncertainty

Live Test Demonstrations (LTDs), or Operational Capability Demonstrations (OCDS) are expensive, time consuming, and difficult for both the Government and the bidders. They are also essential in

some procurements. At times, they are necessary and at other times, they are superfluous. When large, complex systems integration or development programs are to be undertaken, it may be necessary to require an LTD. In these cases, it is important to be certain that the bidders' solutions, their methodologies, and their capabilities are closely examined in a real life scenario simulating the solution offered.

The capacity of a solution to perform at a given level may need to be tested, especially if the solution is offering new state-of-the-art products or are proposing products to be used for the first time. It may also be necessary to have the bidder demonstrate the corporate resources available to respond to unexpected events and schedule and quantity changes. Therefore, an LTD is a way of legitimately evaluating and screening bidders' capabilities to perform.

Some LTD's may not be necessary or offer increased evaluation knowledge. This scenario is most often the case when a procurement asks for an LTD for simple commercial-off-the-shelf (COTS) products. In this case, the technical specifications may be so well advertised and known to the IT industry for these products that the LTD delivers no added-value to the procurement or evaluation. It will, however, add substantial costs to both Government and industry. In other, simpler words, require an LTD only when one is necessary and can substantially reduce program risk.

17) Be certain that COTS and NDI products exist when you call for them

The requirement for COTS and NDI products is a very cost-effective movement within Federal Government procurements. Sometimes, however, the Government asks for products as COTS or NDI items that are not currently available to the general market in this manner. This may be because of misunderstanding on the part of the Government, overstatement on the part of vendors, the failure of vendors to offer these desired products by the time the RFP is issued, or simply because they are not available.

As vendors demonstrate products or discuss capabilities and features, COTS may be an assumed status. Many times these products are then specified in the RFP; and it may be impossible for the bidders to provide these as COTS or NDI, although they are in the development labs or offered to specific markets. In these cases, the bidders spend a considerable amount of time trying to get the products offered or

responding to the Government that the products can not be supplied. Likewise, the time and expense to the Government in addressing or defending their understanding of the availability of these COTS or NDI products to the bidders is considerable.

18) Integrate the debriefing requirements early into the acquisition process

The acquisition process should produce two products: a selection and debriefing for the unsuccessful bidders. The debriefing activity should have two objectives: to tell the unsuccessful bidders why they lost, and to avoid a protest. To maximize the effectiveness of the debriefing activity, the acquisition team should plan for it as an integral part of the acquisition process, beginning early.

Too often, an acquisition team prepares for the debriefing late in the evaluation process, when it is completed. Preparation for a debriefing should begin in parallel with preparation for an award; that is, both should be considered, and given equal weight, when the evaluation criteria are prepared for RFP Section M. Work by the evaluation team should be captured as the evaluation process proceeds, for both the selection and the debriefings.

The same level of diligence and the same considerations should go into selecting a bid as in debriefing one. The criteria apply equally, the evaluation methodology is conducted equally, and the results are weighed equally. The evaluation criteria are used both to make a selection and as the basis for the debriefing. (In fact, if criteria other than those in Section M are used for either, a protest is invited.) The first and most effective line of defense against a protest is for the agency to tell the unsuccessful bidders why they lost and why the successful bidder won, *in detail*, against the published evaluation criteria. That objective is facilitated by the approach outlined above.

19) Cut to a small competitive range when possible

Preparing a large IT system proposal is expensive and requires often scarce resources. Most bidders begin the process believing that their solutions will be competitive and expecting to be selected. Disappointment at not being selected is natural. However, all bidders would prefer to be eliminated from the competition immediately following the determination by the evaluation team that their proposal will not be selected. Better to cut your losses and redirect your efforts than to waste another unit of a scarce resource.

The agency also benefits significantly from a cut in any procurement involving four or more bidders because they, too, can save scarce resources, concentrate on the most qualified bidders, and shorten the evaluation period. Unfortunately this process is not often chosen. Either the agency fears an immediate protest and attendant disruption, or it is not prepared to make and defend a cut.

The best approach is to plan from the start to be able to cut to a competitive range in the event two conditions are met: a sufficient number of bids are received, and rankings among the bids are sufficiently different. The acquisition strategy should incorporate the plan. Bidders should be informed of the intent. Evaluation criteria should be sufficiently detailed to support comprehensive debriefings. Senior management should be informed and ready to support.

20) Have oral presentations and discussions

There are two major benefits in holding oral presentations and discussions. These are:

- Potential to better understand the offered proposals, and
- Opportunity to meet face-to-face.

It is often difficult to understand the full capability or advantages of a proposal by only reading the submitted document. With a full oral briefing of the document and the solution, you have the opportunity to have more in-depth discussions of the details of a solution and raise questions about specific areas, without waiting for time-consuming written questions and answers. These presentations and discussions can be followed with written exchanges to document the sessions.

Orals and discussions give you the opportunity to see your potential providers face-to-face before you award a contract to them. You have the chance to see them in action, how well they interface with you, how they respond to issues, how they may address problems after award, and how comfortable you are going to feel with them. Granted, much of this is subjective, but since you will work with the winner for several years, the process will be of value. It will give you the opportunity to meet the key players and some of the executives responsible for the program from the bidders' corporations. Further, if you combine the presentations with site visits, you can view the corporate capabilities and the depth of personnel available to be called upon to assist in your mission.

Oral presentations give the bidders an occasion to discuss their understanding of the important issues and the mission of the program office. It may also be an opportunity for you to meet the entire bidding team, the prime, and the major subcontractors and teaming partners, to investigate the strength of the entire team, and to view their commitment to the program.

21) Schedule submission of the Cost Volume at least 2 weeks after the other Volumes — more if no BAFO

Bidders prepare their proposal sections and volumes at different paces. Commonly, they conduct trade studies, make architecture decisions, and complete designs prior to proceeding with other steps. Sometimes multiple iterations among options are required. Cost estimating necessarily follows solution definition. Proposal teams lay out detailed schedules to manage the completion of design work, pricing and estimating, and incorporation of the work into text and graphics.

Necessarily, completion of the cost estimates, supporting rationale, and the incorporation of these data into the Cost Volume, must await the completion of all other portions of the work. Hence, bidders set interim (internal) completion dates for non-cost elements to allow sufficient time to complete the Cost Volume. Pressures to perfect the solution, sometimes incorporating late-arriving information (sometimes from the agency) makes it difficult for the bidder to hold to their dates for completion of the non-cost volumes sufficiently in advance of proposal due date to allow time for orderly assembly of the Cost Volume. Even though this may be "*the bidder's problem*," if the agency can significantly mitigate the problem (and it can), all parties benefit.

The agency that recognizes the reality of this problem serves itself by scheduling the submission of the Cost Volume at least two weeks *after* the other volumes. Invariably, the product will be significantly better. It will contain fewer errors and be easier to review. It may represent a lower price if the additional time allows bidders more opportunity to negotiate prices with subcontractors based on the completed solution.

In the special case where an agency considers awarding on the basis of initial submittals, this approach is especially important. It may even be the key to making the no-BAFO strategy work. The reason is that to award without BAFO, the agency must forego the opportunity

for discussions and revisions to proposals. Hence, the more complete and accurate the initial (only) proposal, the better for the agency. Allowing even greater time in this case warrants consideration. The best approach is to ask the bidders how much time they feel is required, during the DRFP period, decide, and stick with that decision.

22) Find a way to waive Cost or Pricing Data requirements

What are the real requirements for cost or pricing data? The Federal Acquisition Streamlining Act (FASA) amended the Truth in Negotiations Act (TRNA) to reduce the requirements for cost or pricing data. If a specific exemption applies, the contracting officer no longer has the discretion whether to require the data. The threshold for submission in civilian agencies has been raised to $500,000 and DoD's temporary threshold of $500,000 has been made permanent. Data requirements have also been relaxed under the rules for acquisitions of commercial items.

23) Critically examine the need for 3rd level and lower detail in the Cost Volume (the lowest value-added exercise of all)

There is typically more cost information required in a proposal submission than can be read or verified in any reasonable amount of time. Bidders are asked to supply this information in both paper and electronic form. Because of the detail desired, and the concern to know all the possible information about the costs, there are instances where following the requirements stated in the RFP would result in 30,000-plus pages of cost information. This volume of data could never be utilized. In addition to increasing the cost of the proposal efforts and the Government's cost to attempt to verify the data, the process adds measurably to the procurement duration. Since we are developing information that will not be utilized, we are paying for a non-value-added activity.

Finally, with greater detail, more errors are inevitable because of the increased complexity of cost element reconciliation and the "*time crunch*" that occurs as the Cost Volume is assembled during the final days of the process.

24) Allow enough time for the intense activities

After "*Lack of Communications*," the second most common complaint heard from bidders concerns time extensions to proposal due dates. Bidders, in general, prefer for the agency to allow enough time for proposal preparation, and then to stick to their schedule. The practice of allocating insufficient time for proposal preparation, then granting an extension is highly disruptive to responsible bidders. Too many acquisition teams employ a strategy of allowing insufficient time knowing that an extension will be "*necessary*" and intend to grant one or more. Of course, bidders bear a large share of the blame because it is frequently they who demand the extension. However, extensions punish the bidders who take the schedule seriously and reward those who do not, and it is the agency who grants or refuses the request. The preferred scenario would be to allow sufficient time for the preparation, and provide for no extensions, except when a major, program-threatening flaw is detected. Just say "*No!*"

The problem is that a complex proposal can require four or more months for assembly, and agencies have difficulties accepting such a lengthy period. The best approach is to ask bidders how much time they will require as part of the DRFP process and allow a little more. Adherence to the schedule also requires preparation and discipline on the part of the agency. The agency must avoid providing grounds for an extension by avoiding RFP amendments. Easily said, of course, but it is frequently possible to avoid material amendments if the DRFP process is conducted effectively. Effective DRFP processes can produce well integrated RFP's and no surprises for the bidder community.

Discussions with agency acquisition people consistently have shown a general lack of appreciation for the complexity and amount of time required for BAFO preparation. They frequently do not understand why four to six weeks or more are required. "*You knew that Call For BAFO was imminent; why weren't you prepared?*" Two conditions cause this situation. The first is that BAFO information is the most sensitive data that a bidder will ever collect. The second is that until Call For BAFO actually is given, the date may slide. Furthermore, even if the bidders are prepared, the process still requires considerable time.

The bidder's BAFO assembly process involves the incorporation of any changes required or allowed into the proposal and the repricing of the proposal. Repricing requires revised estimates by the prime bidder

and solicitation of estimates and quotes from suppliers, frequently multi-tier. Estimates and quotes are revisited to negotiate the most effective distribution of risk and to obtain the most competitive overall life cycle price. Multitier corporate approvals may be required and several iterations may be necessary to obtain the final price.

BAFO prices represent the best proposal that the bidder can assemble. Because of the lengthy acquisition cycle, original proposal details can change materially. In a highly competitive environment, details of competitors' solutions cannot be protected for extended periods because of the mobility of suppliers and the number of different relationships formed and dissolved as bidders form other teams on other competitions, and people change assignments. Accordingly, bidders avoid collecting final pricing data prematurely.

25) Never, never, never slide the schedule

When this statement is made in discussions with acquisition teams, the reaction is usually strong and emotional. However, when case studies of procurements (some very complex), that were completed on reasonable schedules without slides are discussed, certain patterns emerge. No one can guarantee a process for adhering to an IT acquisition schedule, however one deceptively simple observation can be made: "*The way to stay on schedule is to never let it slide.*"

The recipe for success contains ingredients of varying difficulty to obtain. First, the schedule must be reasonable. The portions requiring bidder participation (especially proposal and BAFO preparation) should be established in consultation with prospective bidders. The DRFP period is optimal for establishing these dates. Analogously, time periods for internal acquisition team work elements must be negotiated carefully. The schedule should be published and emphasized frequently and emphatically. Top management support should be pre-arranged and should be prepared to overrule or waive challenges and back the acquisition team. Cutoff dates for internal and bidder interfaces should be set and enforced. Most importantly, no substantive amendments should be issued; they invite schedule slides, and for legitimate reasons.

How can an acquisition team conduct a procurement without issuing substantive amendments? By developing an RFP that is internally and externally well integrated and that, by effective use of the DRFP, contains no surprises for the bidder community. Hence, one that requires no substantive amendments. Finally, when an acquisition

requires no substantive amendments. Finally, when an acquisition team overcomes serious threats to its schedule a few times, everyone else (including the bidders) begins to take the schedule seriously. No one wants to be the cause of compromising it (and bidders can't afford to take a chance).

26) Carefully analyze, map, and monitor all of the stakeholders

The Federal IT acquisition process contrasts sharply with commercial processes in several ways. The most dramatic is the lengthy duration. Next, is probably the large number of players on the Government side. For various reasons, some having to do with protecting the public trust, authority is dispersed and shared among a number of players. Identifying the distribution of authority and understanding people's objectives and needs is critical to success in any endeavor, and certainly so in an IT acquisition.

All organizations have both formal and informal structures. An IT acquisition has, at a minimum, its leader, technical support, contract management, the end-user community, and senior management. In addition, other agencies and elements of the Executive Branch, Congress, associations, and individuals may be directly or indirectly affected by the procurement.

Some of these people have formal roles and are easily identified, others are not. Various forces build and subside during the annual political cycles as budgets and missions are scrutinized. Temporary coalitions arise and fade. People, missions, and external pressures change during a lengthy acquisition. Any large successful endeavor requires strategies to deal with all of these factors. Strategies must be sufficiently robust and agile to recognize and address threats and react appropriately. Deliberate mapping and analysis of the entire community of stakeholders are essential to success.

27) Challenge the unproductive things that your culture requires. Most are NOT requirements of the FAR ("Most of the provisions of Federal Acquisition Streamlining Act (FASA) were not prohibited by the FAR")

Notwithstanding changes to laws and regulations, both Government and industry recognize that cultural changes must take place in parallel. Many people advocate that the Government lead this change. It must accompany the regulatory changes for procurement reform to succeed. OFPP issued new guidelines to encourage

judgment, and plain common sense. In one specific instance, OFPP reversed a long-standing belief in the acquisition community that anything not specifically addressed in the Federal Acquisition Regulation (FAR) is prohibited. The new focus is on what is in the best interests of the Government. Items to watch for (that may be considered nonproductive) include requiring standards that are not applicable, requiring documentation that will not be used or will be superfluous, and requiring too finely-detailed cost information.

28) Objectively evaluate and share acquisition and program successes

IT system acquisition is so complex that when the successful bidder is finally awarded the contract and given their notice to proceed, a great feeling of accomplishment pervades. Regrettably, a successful acquisition does not assure that a successful system will be fielded. Similarly, a troubled procurement does not doom the system being acquired to failure.

Curiously, in the Federal Government community, and in dramatic contrast to private industry, far more attention and scrutiny are typically given to the conduct of the acquisition than to the subsequent performance of the system itself. It is relatively rare for the successes, failures, and lessons-learned after award to be addressed, unless the system provider encounters major difficulties in fielding the system. One reason for this is that it may be several years after award before the results of new systems and processes can be evaluated. Another, of course, is that favorable news does not "*capture the headlines*." Indeed, several Government and Industry attempts to analyze whether acquisitions substantially met agency missions requirements in recent years have met with mixed success.

This culture is harmful to all. If it could be changed such that successful acquisitions and successful systems were dissected, analyzed, and reported, valuable information should result. For maximum effectiveness and credibility, such analyses probably must be conducted by Government personnel (as opposed to industry) who were not directly involved in the acquisition or system development processes. One approach might be to require agencies to prepare a formal report of the acquisition as a final step in the process and establish a "*clearing house*" or library of lessons-learned.

CONCLUSION

As indicated earlier, the above "*lessons-learned*" were derived from the preparation and conduct of interactive presentations made by the authors and other to more than 1,000 government middle and upper managers during the past 5 years. Accordingly they are the "*intellectual property*" of us all. Many of these practices are now routinely followed by some agencies. Others are under consideration or under trial. Given the great diversity among agencies, there are some government managers who challenge whether some of these practices are allowable. However, we firmly believe that none of the above require changes to anything other than internal agency policy.

We also firmly believe that all of the above practices pass the test of being beneficial to both government and industry. Each item is worthy of consideration by itself, and taken as a whole they aggregate to a partial set of best practices given the current state of the art of IT acquisition management. As we continue to interact, there will be many more good ideas debated among us. The authors hereby solicit any comments in any form (criticism, corrections, additions and the like) from any interested person.

CHAPTER 13
Addendum B

Contracting for Success

Jerome S. Gabig, Jr.

NOTE: You will find this article in Volume 2, Appendix O, *Additional Volume 2 Addenda.*

CHAPTER

14

Managing Software Development

CHAPTER OVERVIEW

In 1975, MITRE and John Hopkins University Applied Physics Laboratory (APL) conducted studies for DoD concerning weapon systems software management. They concluded that the major contributing factor to weapon system software problems was a lack of discipline and engineering rigor applied consistently to software acquisition activities. They also concluded that it is essential to require the development contractor to apply a highly disciplined set of engineering practices to the detailed design and implementation phases of development. Once your contractor is onboard, you must ensure a disciplined, controlled, engineering process is established and maintained. The most successful management approach is one that is founded on the principles of continuous process improvement where quality is the goal. Quality is achievable through a quality process, hands-on management, and by following the guidance provided in this chapter which is based on lessons-learned DoD and industry-wide.

*By making **quality** the number one priority, all other elements of success fall in line. This entails championing a **process-approach** to software development where reassessment of methods, procedures, tools, and products is a common activity. Quality is defined as the degree of excellence in your product. Product quality and process quality are, however, interdependent and inextricably linked to one another. Product excellence cannot be achieved unless a quality process is in place to development it. This can only achieved through strict adherence to **software engineering discipline**.*

*Statistically sound management techniques transform the software process into an engineering process. Engineering discipline must be embraced from the total life cycle, total systems perspective. Separating hardware from software violates the most fundamental principles of systems engineering and can only increase overall program risk. A **systems perspective** focuses on producing engineered software that is reliable, maintainable, efficient, timely, and affordable. This is accomplished by concentration on **error prevention** (rather than correction) through a defined process, cohesive teamwork and communications, control of procedures, and satisfaction of user needs.*

CHAPTER 14 Managing Software Development

The biggest development issue (and the phase requiring the largest commitment of resources) is the definition and analysis of ***user requirements****. This area requires strong, decisive leadership to be successful. Requirements must be scrubbed to only those that are essential. Planning for evolutionary, incremental requirement upgrades is the most efficient method for controlling requirements creep. The VHDL hardware design language simulates hardware requirements (primarily for embedded processors) and reduces the risk of inadequate hardware selection before all software requirements are established.*

One approach for ensuring successful requirement validation is ***prototyping,*** *which gives users a touch-and-feel for how the system will perform once built. Another area requiring your attention is software* ***design****, where building quality in is the goal. The best way to guarantee quality software is through its* ***architecture****. A quality architecture is one that is flexible, allows for evolutionary change of the system, and based on approved architectural standards.*

Although quality cannot be tested into software, testing is an important quality control activity. ***Testing*** *for defects early on is an important process improvement tool. It enables cost-effective defect resolution, identification, and removal through a self-correcting development process that is stable, modeled, measured, and predictable. Efficient testing engineers quality into software fast — at reduced cost.*

Technical ***documentation*** *helps users and maintainers in the operation and support of your product. Software documentation is a large determinant of product quality. It is also a major factor in support success. Be careful not to overload your development effort with unnecessary, costly-to-produce documentation requirements. Trim the documentation fat where possible, and only produce that which is essential for engineering, configuration management, and PDSS needs.*

CHAPTER 14

Managing Software Development

WINNING THE BATTLE WITH QUALITY

In 1732, Field Marshall Maurice Comte de Saxe, victor of the Battle of Fontenoy and one of the most innovative soldiers of his day, made a perceptive statement that applies to winning in today's software arena:

> *It is not the big armies that win battles; it is the **good** ones.* [deSAXE32]

Like the good army, it is the **good** software that wins the software management battle. How to manage the development of ***good software*** is the focus of this chapter.

In Chapter 1, *Software Acquisition Overview*, you were told that the most common problem cited in major software-intensive acquisitions has been ***poor management***. Most DoD programs have been managed by cost and schedule — not the **quality** of deliverables. When the pressure of meeting schedules and reducing costs intensifies, quality is sacrificed. To accelerate a late product, the first thing managers do is to cut back on those activities not absolutely essential for product delivery — usually verification and testing. This always results in diminished product quality. [GLASS92]

Too often, program managers are not process-oriented and do cultivate a process to control the effort. As you learned in Chapter 12, *Strategic Planning*, even with the best process in the world, if you have not planned for sufficient schedule time and money to build quality into your product, your program is doomed. It follows that to avoid

repeating past failures, you must pursue an alternative management style that improves upon the cost/schedule paradigm. The equation for software success must change from:

Software Product = On Time + Within Cost

Quality must be factored into the equation as ***the driving factor*** for all software development management. The equation for software success must be:

Software Product = Quality + On Time + Within Cost
[GLASS92]

[Refer to Chapter 8, Measurement and Metrics, for a definition of software quality.]

Walt Disney expressed this concept of quality when he said,

> *I don't worry whether something is cheap or expensive. I only worry if it is good. If it is good enough, the public will pay you back for it.* [DISNEY76]

Requirements definition adequacy, the degree of software engineering discipline applied during design, and the quality of the development process are all important to the production and maintenance of quality software. Emphasizing that quality must be ***built-in***, rather than tested into the product before delivery, must be a management prerequisite. Testing can only asymptomatically reduce the risk of latent defects — but cannot eliminate it. Quality must be built into software requirements and designs by including fault-tolerance and the removal of all failure modes.

SOFTWARE DEVELOPMENT PROCESS

The **software development process** consists of the way by which people, procedures, methods, equipment, and tools are integrated to fulfill user requirements through the production of a high-quality software product. By **standardizing** on a disciplined process and comprehensively defining that process, you will reap the benefits of reduced training requirements, maximized use of available resources, increased productivity, and product predictability.

Statistical management techniques provide the discipline that transforms software development into an engineering process. They are key to producing quality software products that are reliable, maintainable, efficient, timely, and affordable. Whichever practices you include in your management process, a total system/total life cycle approach to building quality software is a must. Examples of the software engineering practices used to build-in quality on the **Space Shuttle** *[and the chapters where they are discussed]* relevant to all major software-intensive systems, include:

- Quality engineering,
- A defined software development process,
- Formal peer inspections *[discussed in Chapter 15, Managing Process Improvement]*,
- Rigorous configuration management *[discussed in Chapter 15, Managing Process Improvement]*,
- Continuous process improvement *[discussed in Chapter 15, Managing Process Improvement]*,
- Statistical process control *[discussed in Chapter 15, Managing Process Improvement]*,
- Defect causal analysis and prevention process *[discussed in Chapter 15, Managing Process Improvement]*,
- Automation of software production processes *[discussed in Chapter 10, Software Tools]*, and
- Quality monitoring metrics and interpretation *[discussed in Chapter 8, Measurement and Metrics]*. [KELLER93]

Systems Perspective

The 1991 final report of the Government/Industry Acquisition Process Review Team, *Clear Accountability in Design (CAID)*, identified significant areas for process improvement in major Air Force acquisitions. These findings are particularly meaningful when employing a **systems perspective**. CAID solutions focused on three opportunities for management process improvement applicable to all major software-intensive systems.

- Design management and review *[reviews and audits are discussed in Chapter 15, Managing Process Improvement]*,
- Risk management *[discussed in Chapter 6, Risk Management]*, and
- Effective teamwork with clear roles.

The report found that all areas for process improvement are based on an underlying set of conclusions:

- It is the Government's responsibility to define requirements and industry's responsibility to design to those requirements;
- Strong focus is needed on demonstration milestones with specific entry/exit criteria;
- System development is an evolutionary process that requires continuous cost, schedule, and performance tradeoffs, especially in requirements, objectives, design, and testing;
- Early and continued emphasis must be placed on effective teamwork between the Government and industry;
- Program teams must focus on sound, upfront risk management using effective abatement techniques and adequate funding; and
- ***A judicious amount of common sense and a willingness to deviate from the norm, when necessary, is essential for quality system development.***

Design Management and Review

The CAID team found that new **operational requirements** developed by using commands become effectively frozen when transferred to the acquisition command. While the acquisition command and defense contractors may be involved early on in influencing systems requirements, the program office and industry perceive that operational requirements are fixed and cannot be violated. Once the system-level specification is placed on contract, the contractor translates requirements into a system design with the Government giving advice at **Preliminary Design Reviews (PDRs)** and **Critical Design Reviews (CDRs)**. Any changes impacting specifications must be processed through the laborious **Engineering Change Proposal (ECP)** process. Government and contractor cost and schedules are increased by inflexible specification management and government design control at the detailed solution level. ***This results in the translation of operational requirements into fixed specifications*** — negating the possibility for alternative solutions. The process does not address system development realities where the design is iteratively refined and matured, reflecting technical tradeoffs.

The CAID report's solution to these DoD-wide acquisition problems was an **evolutionary system development process**. Combined government/industry teams must be involved in influencing and recommending the tradeoffs essential to defining **minimally**

acceptable requirements. These teams must also participate in developing the acquisition strategy at the very beginning of the process.

Effective Teamwork with Clear Roles

The CAID report further recommended that DoD program managers be empowered to lead a **government/industry team** *[known as Integrated Product Teams (IPTs) or Integrated Process Teams (IPTs)]* effort dedicated to balancing technical risk, time, and money. Representatives assigned to the team should be given full authority to represent their parent organization to strengthen the authority of the team and improve the timeliness of decisions. These teams must be formed early, with core team support at Milestone 0 and full-team support at Milestone I and beyond. There must be a continuity of **key personnel**, a commitment to **concurrent engineering**, and **collocation of team members** where practical within Government and industry. The report also recommended that the Government match the contractor's organization where possible — not the other way around.

The team must establish an **evolutionary management process** to achieve and maintain a balanced requirements baseline consistent with tradeoff planning. The team must maintain close and continuous communications and protect against unwarranted direction to contractors through simple and timely *memoranda of understanding (MOU)*. The Government must have timely access to contractor cost, schedule, and technical performance databases, and use as many meaningful progress milestones as possible. The team must also establish common goals and mutual motivations to foster frank, constructive dialogue, recognizing that some government/industry goals and incentives are inherently different. [CAID91]

Process-Approach to Quality

An important factor to always remember when following a development methodology is that no technique should be used without a predefined commitment to ***mold*** that technique to meet your specific program needs. Lessons-learned from the Air Force **Nuclear Mission Planning and Production System (NMPPS)** recommend that software development programs be structured with ***relatively short intermediate steps that give the user something concrete to see and touch***. This approach fulfills several objectives:

(1) it provides quantifiable evidence of program progress; (2) it permits incremental evaluation of the delivered product; and (3) it contributes to increased commitment to the program. [KEENE91] However the process is executed, in whatever sequence, the elements of the process can be defined as follows. The problems often encountered in these activities are listed as areas for process improvement.

- **Requirements**. The recurring problem with requirements is the tendency to state a solution to the problem — rather than strict definition of the problem. If a solution or partial solution is specified, the definition becomes a requirement which can prematurely preclude alternative solutions.
- **Design**. A recurring problem during design is stopping too soon (leading to an insufficient solution) or not stopping soon enough (leading to an implementation phase overlapping with the design phase) — wasting time and money.
- **Implementation**. Problems during implementation include misunderstandings of interfaces and processes. Software applications are made of a myriad of minute details, many of them related and many of them complex systems in their own right. **Defects** due to a lack of understanding, design flaws, and carelessness are a very human by-product during **coding**.
- **Testing**. The major problem with testing is impatience. Testing is the painstaking process of trying out all requirements, all the structural elements, and as many logic path combinations as cost, schedule, and common sense will allow. The temptation here is to stop short, declare the software ready, and ship it off to the users.
- **Maintenance**. The greatest problem during software maintenance is poorly designed, poorly developed code, and inadequate or nonexistent documentation. *[Maintenance is discussed in Chapter 11, Software Support.]* [GLASS92]

Software Development Plan (SDP)

The ability to win the software battle depends on the success of the plan. As Napoleon explained,

> *Nothing succeeds in war except in consequence of a well-prepared plan.* [NAPOLEON08]

The **SDP** (usually submitted in draft form with the offeror's proposal) is the key software document reflecting the overall software development approach. It includes resources, organization, schedules, risk identification and management, requirements management,

supportability, training, open systems and standards compliance, Ada use (or Ada waiver rationale), tools and environments, data rights, metrics, quality assurance, control of nondeliverable computer resources, and identification of COTS, reuse, and GFS the offeror intends to use. SDP quality and attention to detail is a major source selection evaluation criterion. *[Refer to MIL-STD-498 for suggestions on SDP content and requirements.]*

CAUTION! A poorly written SDP is a warning sign! If awarded the contract, an offeror with a deficient SDP has a low probability of successfully completing their obligation to the Government! Be aware also of those offerors who talk a good game. A well-written SDP must be backed up by a high software maturity assessment *[discussed in Chapter 7, Software Development Maturity]* **in addition to other source selection discriminators.**

An **evolutionary development approach** with **incremental delivery** of capability and software prototyping techniques can yield early results and demonstrate the contractor's competence to develop software components accurately and reliably. An early *"warm-and-fuzzy"* about the SDP builds the foundation for the teamwork and disciplined trust vital to life cycle cooperation and success.

Software Development Recommendations

The use of modern software development techniques, CASE tools, and metrics are essential for successful software development and support. You must ensure these are used for all software efforts and provide the guidance and resources for their implementation. Specifically, you can greatly help your program by doing the following:

- Program management by committee is ineffective. A single manager is needed with the rank, background, responsibility, and authority to carry out the program.
- You must actively pursue a commitment to quality from your industry/government development team. Quality can be implemented through training and by formally establishing quality objectives within your software requirements specification. This will increase the importance and visibility of software quality by providing clear, measurable goals. The cost of quality will be balanced with improved product performance, cost, and schedule.

- Make sure all stakeholders (users, developers, testers, and maintainers) participate jointly in requirements definition and analysis, and that they are mutually responsible for ensuring requirements are clearly documented, implementable, and testable.
- Make sure requirements documents clearly define the quantifiable attributes of quality code.
- Use **COTS** and reusable components when appropriate. However, be aware of associated data rights issues that may affect supportability.
- Mandate maximum use of modern development practices, such as object-oriented analysis and design, during early program phases.
- Require that contractors use an appropriate set of CASE tools and acquire CASE tools for in-house use in progress tracking. Ideally, each member of the contractor team, including all subcontractors, will use the same CASE toolset/software engineering environment (SEE). Where all team members have the same toolset, the opportunity for integration problems is greatly reduced.
- Encourage evolutionary introduction of standard CASE tools for all subprocesses including development, testing, and maintenance. However, purchase, train for, and employ only those tools for which corresponding processes have been defined.
- Allow adequate time to learn and gain experience with both the method employed by the CASE tool and the tool itself. *[Consult STSC tools documents and the I-CASE program office at Gunter AFB for further guidance.]*
- ***Do not modify COTS***. Use them for their intended purposes in their intended environment.
- Preplan replacement of COTS items. Include commercial-item support requirements in your procurements.
- Put architectural requirements in software specifications.
- Emphasize early satisfaction of architectural requirements before a large investment in design and coding of applications software.
- Incentivize contractors to reuse architectural concepts that are proven to be effective.
- Make sure architecture configuration is controlled throughout the life cycle.
- Maximize the use of Ada and COTS tools.

Lessons-Learned from SSC and CSC

- Understanding the development approach and methodology used by the developer makes it easier for all team members to stay focused on the task, reduces rework, and eliminates confusion over changes in the process. The development methodology should be documented in the SDP and refined throughout the development life cycle based on lessons-learned.
- A standardized representation of the problem domain (through the use of a graphical OOA tool) helps functional users and systems analysts obtain a common understanding of the system's functionality and requirements.
- Several steps must be taken prior to a site visit to define user requirements: (1) form small 2-3 person teams; (2) obtain and review organizational charts and background material for the site to be visited; (3) review problem domain policies and regulations; (4) identify site POCs and phone numbers; and (5) establish the objectives and parameters for the visit.
- Coordinate the requirements definition visit with the OPR, and if that person must do the interview scheduling, emphasize the necessity for at least an hour break between each interview.
- Appoint a team coordinator for each requirements definition visit. The coordinator's responsibilities should include: appointment verification; team assignments to meet schedule changes; conducting after-hour reviews and recaps of the day's activities with other team members; maintaining focus on the visit's objectives; and ensuring gray areas are resolved before the team leaves the site.
- During systems requirements review, place more emphasis on reviewing the systems requirements and interfaces, than on program methodology and metrics.
- On requirements definition trips, it is helpful to talk to groups that interface with the domain being modernized to clarify or uncover requirements. Include supported functions or agency interviews as part of the requirements definition process.
- The systems analyst must work closely with the users and other members of the requirements analysis team. During systems analysis, functional users should be allowed to pursue breadth and depth of analysis as ideas occur to them. In addition to facilitating the flow of ideas, the analyst must ensure the analysis continues in a timely manner. Analysis must not get bogged down in details which can best be addressed in a subsequent development phase.
- Rewrite the SSR section of the SDP and add the requirement to present information on the scope and methods used during requirements definition.

- Using a generic term before applying technical solutions can cause confusion and miscommunications. POSIX should not be referred to as a single standard when it actually represents a family of standards. The definition for POSIX compliance is: (1) an application is compliant if all of the services it uses are part of the approved platform/set of standard services; (2) a platform is compliant if it supplies all the services required by a standard; and (3) in the event of an overlap, the application must choose between services. (Ada LRM services are preferred.)
- The function point count of an OOA model can inflate inherited attributes and services; therefore, the counting methodology should be refined to give a clearer representation of the functionality of object-oriented systems.
- As an OOA model grows in size and complexity, a configuration management function should be added to the analysis model. Because a minor change made in an object can cause catastrophic changes in the model, all but one team member should have read-only permissions with its use. If changes to the drawing or underlying documentation are needed, the files can be released (write permissions temporarily granted) while changes are made.
- Because problem domain issues come up during team reviews, functional user participation on review teams provides quick and efficient closure to these issues.
- Training on the reusable software architecture is required for the entire design team.
- Select a modeling methodology that provides a clear, single representation of the system as it evolves throughout the development process. Multiple model representations create confusion and increase risk.
- To eliminate confusion on where to obtain edits for a data element (the SRS or the data dictionary), metadata should be captured in a single repository. A data dictionary team (comprised of analysts, data dictionary experts, and dictionary users) should be established to coordinate all data dictionary efforts.
- Changes to the design model and documentation impact on the generation of the SRS; therefore, a technical review team should review the documentation produced by the model prior to porting the documentation for formal delivery.
- Unfamiliarity with data dictionary naming conventions unfavorably impacts SRS generation and the approval of data elements.
- Lack of designers during the requirements analysis phase produces gaps between the analysis and design; therefore, designers should be included on the requirements analysis team.

- A prototype should be built along with the requirements analysis model so that prototypers can ask the analysts what each software function should perform and whether or not the placement of functions is accurate.
- Ensure tailoring of the DID used to generate the SRS is finalized before work on the requirements analysis model begins.
- Maintaining and posting requirements analysis changes reduces unnecessary prototyping of obsolete *class-and-objects* and avoids confusion as to what has been prototyped and what has not.
- The screen design group should consist of a team leader, a scribe, and technical representatives from the task (less than 7 members is recommended). A flexible screen design that reflects user input makes transition to the final screens easier.
- To increase prototyping efficiency, prototyping should start as the requirements analysis model is being built. Objects should be coded as they are created, so that functions (services) can be attacked early. To reduce the risk of prototyping objects and services that may change, the systems analysts must provide estimates of stability. Prototypers should only implement the most stable objects first.
- Prototypers must be involved in reviewing software requirements, as they bring needed expertise. Traditionally, this has not been the case. If prototypers still have work remaining when the requirements review occurs, they should suspend that work, because the final requirements will impact whatever prototyping occurs.
- A clear representation of the problem domain is essential for the understanding of and communication about the system. The notations in the Coad-Yourdon methodology graphically represent generalization specialization structures and the attributes required in each part of the structure. [COAD90] *[The Booch notation is very fuzzy on atomic level versus non-atomic level classes.]* A single consistent representation is needed across all phases of the life cycle (i.e., a single model).
- Screen design rules development is time-intensive. It takes time to role play the functionality of each screen. Use information captured prior to screen design to speed up the process. Rules for functionality should be added to the design folders and used during screen design.
- Alternate screen design layouts, based on human engineering considerations, should be introduced early in the design process and presented to the users for their consideration.
- The Government Technical Review stands alone as it precedes all major reviews. This repeatable process should be included in the SDP.

- A "*Phase Kick-off*" meeting should be held prior to each phase of development. This ensures that team members understand the major activities and deliverables associated with the current phase. The meeting should address the near-term schedule, planned activities, and deliverables. Representatives from recently completed (or still in process) task(s) should be invited to share their lessons-learned.
- Internal Review (IR) screen design meetings should not devote time to designing or redesigning screens. The package should be reviewed by the assigned reviewers and their objections and comments presented and discussed. The screen design team submitting the screen package should conduct a pre-IR to review the package and incorporate objections or comments from within the team. This ensures the package submitted to the reviewers has been informally pre-reviewed and accepted by the team—reducing the actual time necessary for the IR meeting.
- The IDEF0 model is difficult to maintain given the evolutionary nature of the software development process. Updating the IDEF0 model is often neglected and process improvement becomes based on process analysis charts. The purchase of additional copies of the tool, Design/IDEF, helps to distribute this maintenance load.
- A set of standards should be adopted before a screen is baselined.
- The lack of coding guidelines can cause considerable revision of software considered complete. Therefore, use a set of checklists for software coding style and content using the *Ada Quality and Style Guide* from the Software Productivity Consortium.
- Formal guidance must be documented and guidelines in place prior to implementing data standardization.
- Ensure the data standardization team consists of people with appropriate levels of technical, functional, and corporate knowledge.
- Document and inform all data standardization team members on generic element naming guidelines to ensure modifiers are used.
- Define the process for submitting data elements in the SDP. The timing should be no later than the point at which the task's subject areas have been reasonably stabilized.
- Ada training needs to be thorough with a hands-on approach to using the analysis model produced by the Software Architecture Group.
- A detailed document on what is needed and available in the way of hardware/software resources should be made at the onset of each new task to mitigate potential risks in meeting deadlines.
- Use the documentation capabilities of the Ada design model. Add documentation to the description of the top-level architecture.

- Consider the look, feel, and design of all the other architecture components when designing a new component. Move on with design and Ada specifications before implementing Ada bodies.
- The amount of time given to design must be enough to fully mitigate potential risks. For example, changes to the design that are implemented after portions of the design have been coded require significant time and effort better spent in upfront design. Therefore, it is recommended that the design on the **OOD model** (architectural design) be complete across all components of the top-level architecture before coding. Detailed design should then proceed incrementally to mitigate risk.

NOTE: See Chapter 5, *Ada The Enabling Technology*, for a description of the programs upon which these lessons-learned are based. See also Volume 2, Appendix O, *Additional Volume 1 Addenda*, Chapter 14, Addendum C, "*On-Board Software for the Boeing 777*."

SOFTWARE REQUIREMENTS

Paul Paulson, president of Doyle, Dane and Bernbach, a large New York brokerage firm, was quoted in the *New York Times* as saying,

> *You can learn a lot from the client. Some 70% doesn't matter, but that 30% will kill you.* [PAULSON79]

You must approach the requirements task with strong leadership that emphasizes risk reduction through **evolutionary development** *[discussed in Chapter 3, System Life Cycle and Methodologies]* and **prototyping** to ensure quality issues are translated into functional requirements. The software system must, in addition, be analyzed within its ***environmental framework***. This analysis may be performed in accordance with one, or several, **structured analysis techniques** (such as **functional decomposition**, hierarchy diagrams, object-oriented analysis, data flow analysis, or state transition charts). Methods include: **object-oriented** (data-oriented) [COAD90], **process-oriented** (functional or structured analysis) [YOURDON90], and **behavior-oriented** (temporal, state-oriented, or dynamic; e.g., essential systems analysis) [McMENAMIN84]. Each of these techniques view the system being developed from a different perspective.

The approach selected by the development team depends on the type of software system being defined, and the approach that most clearly states the problem. Further analysis involving user scenarios, transaction modeling, performance modeling, and consistency checking among viewpoints must also be performed. This ensures overall requirements consistency. Requirements so derived must then be validated with the users prior to development to guarantee that the system can, and will in fact, be built. **Validation approaches** include performance modeling and prototyping of those software components deemed critical to software success. Another good litmus test for the validity of a requirements package, used on the **F-22 Program**, is to check whether designers from two different development team organizations have identical understanding of a set of bottom-level elements in the requirements hierarchy. If not, your team process is flawed, and the chances the pieces produced by various team members will integrate smoothly is close to zero.

Requirements must be clearly documented and implementable. They must also be well-stated so they are easily understood by the designer and programmer. One measure of clarity is that requirements must be ***testable***. If requirements are satisfied, you should be able to quantitatively test them. Classic examples of untestable requirements are those that state the system must be *"user friendly"* and provide *"rapid response."* Such requirements are meaningless to the designer, and are a potential source of endless arguments when the software is delivered.

A description of **requirements tests** (or measures) must be included in the specification. T**esting** must demonstrate that, if successfully completed, the delivered software will satisfy the requirement. The need for testable requirements demands that testing issues are addressed early in the program. Software **test personnel** (in addition to the developers, users, and maintainers) must take an active role in the requirements definition, analysis, and software design phases. The formal assessment of **quality objectives** should be an integral part of this effort. Unless a user need is correctly and completely stated, it is unlikely that either quality code will be written or a test can be performed to determine if the software satisfies the need and quality requirements *[e.g., quality attributes such as those listed in Chapter 8, Measurement and Metrics, Table 8-4.]*

Many techniques have been developed to assist in specifying and documenting requirements, such as IDEF and CASE tools *[discussed*

in Chapter 10, Software Tools]. Whatever the tools or methods used, the analysis should include a basic series of requirements activities.

- If requirements are uncertain, build a prototype or model the information domain,
- Create a behavioral model that defines the process and control specializations,
- Define performance, constraints, and validation criteria,
- The SRS must be written or depicted, and
- Conduct regular formal technical reviews. [PRESSMAN93]

Figure 14-1 illustrates the requirements definition and analysis process performed for the **F-22**. A joint relationship among all stakeholders must continue throughout development. Eventually, this effort will result in documentation or data that directly ***cross-references test cases to requirements and code***. At the same time, developers and testers should independently plan, design, develop, inspect, execute, and analyze software test results.

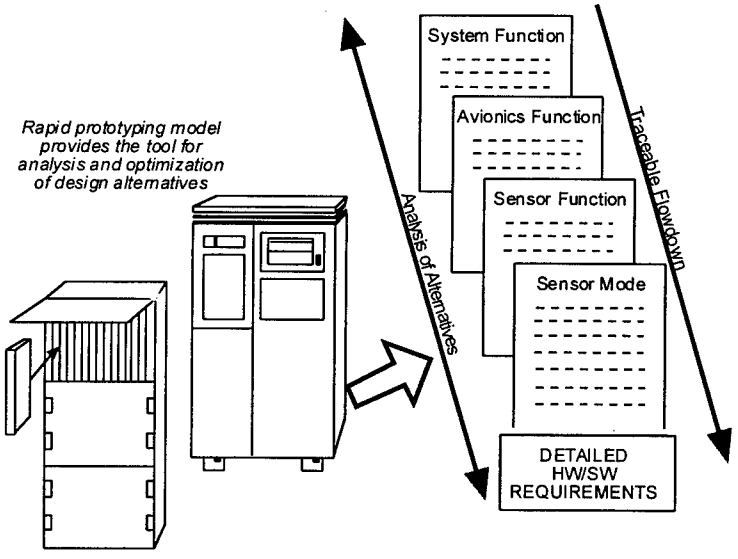


Figure 14-1 F-22 Requirements Process

NOTE: See Addendum B of this chapter, "*If Architects Had to Work Like Programmers.*"

Software Requirements Specification (SRS)

The successful completion of the requirements phase results in a **Software Requirements Specification (SRS)**. Common sense must be used when writing these specifications so that they are realistic, achievable, and not just *bells-and-whistles*. It must be kept simple and short. **Quality attributes** *[discussed in Chapter 8, Measurement and Metrics]* should be defined such that the designer knows how to go about achieving them, and the user knows whether they have been achieved when the software is delivered. [GLASS92] Remember, ***quality must be testable and measurable***. To achieve this, there must be an open, honest, and cooperative free exchange of information between the Government and the developer (contractor) as reflected in the SRS.

The specification process is one of ***representation***. Requirements must be represented in such a way as to facilitate their successful implementation into software. The characteristics of a good specification are:

- Functionality is separated from the implementation. Specifications must be expressed entirely in the *"what"* form, rather than the *"how."*
- The specification language must be process-oriented—the process to be automated and the environment in which it is to function and interact must be defined.
- The specification must describe the software within the context of the entire system.
- The specification must be an empirical representation, rather than a design or implementation representation, of the system.
- A specification must be comprehensive and formal enough to determine if the implementation fulfills randomly selected test case requirements.
- The specification must be flexible, enhanceable, and never totally complete.
- The specification must be localized, loosely coupled, and dynamic. [PRESSMAN92]

Dobbins claims that as long as developers insist on writing software requirements in prose form, requirements will continue to be the source of expensive software defects. He recommends the acquisition and use of one or more of the emerging specification generation techniques, many of which require the use of CASE tools. The tools

selected should be based on *ease-of-use* and the ability to perform comprehensive real-time analysis and evaluation of the requirements package as it is being developed. [DOBBINS92]

Interface Requirements Specification (IRS)

Never lose sight of the fact that hardware and software development are intimately related. Although they are developed in unison, for major programs, software is always on the system's **critical path**. Early consideration of how the software is to interface with the system and other software is necessary to achieve the benefits of cohesive, **interoperable** systems. Proper **integration** of hardware and software can be assured through carefully identified **interface requirements** and prudently planned reviews, audits, and peer inspections *[discussed in Chapter 15, Managing Process Improvement]*. Such systems provide improved accuracy, currency, and quality. Early identification of integration and interface requirements also prevents **redundancy**.

Software interface requirements are documented in the IRS. In complex system developments, with multiple developers, each contractor must have a baselined IRS to ensure interface discipline. There must also be a requirement that each contractor's software system interface with other designated systems. Otherwise, each contractor can change their interface at will, affecting other contractors' efforts. Not baselining **Interface Control Documents (ICDs)** also gives contractors a mechanism to shift the schedule continuously. While ICD changes can lead to additional expense, uncontrolled change is even more dangerous. A more acceptable method is to develop to a given version of an ICD while still having the contractor maintain and update that ICD. The contractor then assumes the responsibility of maintaining current ICDs and of meeting the requirement. Program milestones should be used to determine which ICD is being used to develop any given phase of the system.

NOTE: Refer to Requirements Determination Process by EDS. See Volume 2, Appendix E for information on how to obtain a copy.

Requirements Management

Management of technical requirements is the most important, and often overlooked, software management issue. Ideally, requirements are fully identified before the SOW is written. In practice, this is almost never the case. In large, complex software-intensive systems, requirements continually evolve throughout the system's life. Therefore, they must constantly be managed because they significantly impact total system development cost and schedule. Requirements creep often occurs on long procurements. The users have time to see the possibility of all the features they can have, or want changes to their original vision of the product. The operational environment changes or technology advances. The software contractor may want to accommodate the user, but through requirements creep, loses control of the product. If your contractor is unable to do so, it is your job to step in, hold the line on requirements, and take control of a situation that can run up cost and schedule at lightning speed. Failure to draw a line in the sand on user requirements can be downright fatal for your program. **Freezing requirements** through firm baselines is essential. It does not, however, make it impossible for the user to make changes. **Evolutionary/incremental buildup** of functionality is possible if it is planned and budgeted to occur at **milestone decision points** where requirements are re-baselined. **Version control** and **tracking**, including updating documentation, are other essential parts of the requirements configuration management task *[discussed in Chapter 15, Managing Process Improvement].* Figure 14-2 illustrates how F-22 requirements were baselined with planned incremental buildups of functionality.

Your contractor's SDP should address their understanding of the **requirements stability issue**, how well they will manage the **requirements change process** and **evolutionary requirements**. [MARCINIAK90] Your contractor's management of requirements must stress a commitment to an iterative process that utilizes structured requirements methods and appropriate tracking and analysis tools. **Traceability** from original, identified needs to their derived requirements, designs, and implementations must be assured. This process must also identify major stakeholders (or viewpoints) of the system. You must ensure that individual stakeholder needs are consistently collected, analyzed, and documented. A formalized process must also be used to enfranchise stakeholders through requirements teamwork. A two-way requirements **traceability matrix** should be used to ensure completeness and consistency of requirements among all levels of development (from top-level down

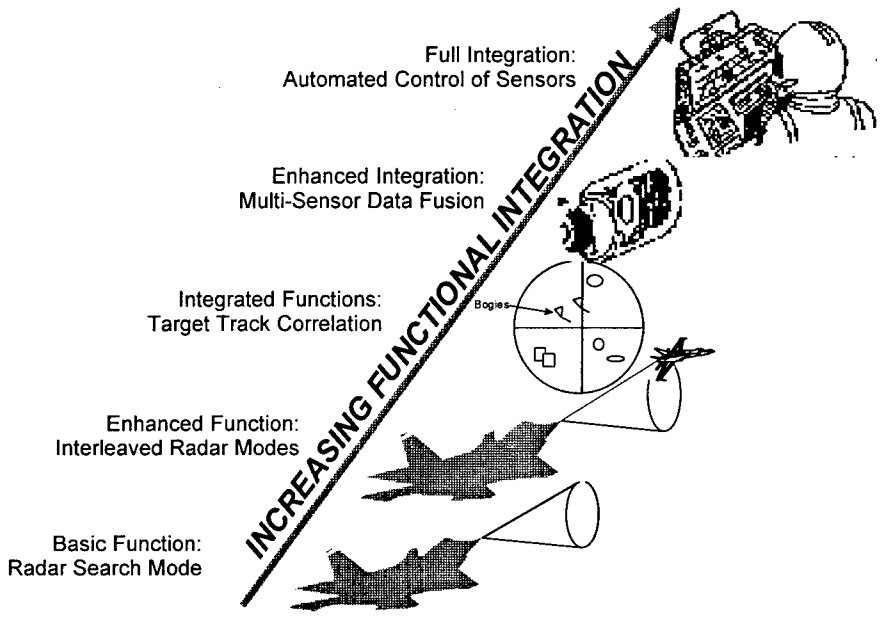


Figure 14-2 F-22 Baselined Requirements with Incremental Buildup

to code-level). [SHUMATE92] *[For help in choosing automated tools for requirements analysis and control refer to the STSC's Requirements Analysis and Design Tools Report, March 1994.]* A **requirements team** includes multiple working groups, such as:

- An Operational Requirements Document Support Team,
- A Program Requirements Team, and
- An Operational Requirements Working Group. [SPAT92]

Prototyping

Prototyping, along with a structured analysis process and performance modeling, is an effective means to evolve and clarify user expectations. They can be used to resolve conflicts among cost, schedule, performance, and supportability; to ensure users and developers have the same understanding; and to **validate requirements**. Prototypes provides a better way for resolving the statement, *"I'll know it when I see it,"* than documenting requirements in English, with all its ambiguities. [KINDL92] Prototypes, which include the results from rapid prototyping techniques, executable models, and quantitative descriptions of behavior (such as structured

prototyping languages or graphical representations), are powerful tools for deriving correct hardware/software partitioning, for performance testing, and for eliminating significant sources of risk. Remember, ***prototypes must be useful, not just demonstrations or models of the system.***

Prototyping involves the early development and exercise of critical software components (e.g., user interfaces, network operating systems, resource managers, and key algorithm processors). They are comprised of the user interface, its interaction details with the proposed system, and executable functional models of critical algorithms. They are different from demonstration systems *[discussed in Chapter 12, Strategic Planning]* which provide usable evolutionary increments. Normally, in MIS and C3 systems, prototypes demonstrate screens and limited functions — ***not actual software that works!***

One method for developing a prototype to is build it from **reusable components**. Because the components already exist, a prototype built from reusable parts is the easiest, cheapest, and quickest to build. It can provide rapid functionality since it is built from previously coded and tested components. Another way to build a working prototype is through a tool such as **UNAS** *[discussed in Chapter 10, Software Tools]* which can generate a demonstrable level of functionality in Ada code. An ability to *plug-in, plug-out* **COTS** products can also greatly speed up the prototyping process. The least desirable prototyping approach is one which uses **HOLs** and/or rapid prototyping tools that only build a *quick-and-dirty* skeleton of the system. While the external facade (e.g., front-end screens with no code behind them) may give the user a *touch-and-feel* for what the final system will be like, there is nothing behind that front-end prototype (i.e., no functionality that the user can execute to determine if it will do something useful). [YOURDON92]

You must make sure your contractor's SDP addresses coordinating prototype development with the **system end-user** to ensure a realistic requirements validation process occurs. As a minimum, the user must review and approve all prototypes of critical components. Reiteration of this process is often necessary to include additional requirements analysis, specification, and validation if the prototyping exercise falls short of user expectations. The approval of the prototype(s) constitutes a **baseline for system requirements** to be incorporated in the **SRS**.

Prototyping Benefits

Major prototyping benefits include: clearer understanding of requirements, particularly if a user-interface prototype is demonstrated; quicker identification of design options and how they may be implemented into code; and resolution of high-risk technical issues in areas where the system may be pushing software state-of-the-art. Prototyping also has a high impact on a certain class of defects and can be used as an effective **defect prevention** technique. Although with large, complex software developments it is usually not possible to derive all the functional requirements upfront, there is evidence that software developments using prototyping tend to reach functional stability quicker than those that do not. With prototyping, on the average, only 10% or less functions are added after the requirements phase; whereas, without prototyping 30% or more functions are added after requirements analysis. This leads to unanticipated and unfunded cost and schedule overruns. Also, defect correction costs associated with these late, rushed functions, exacerbates the problem, as they are more than twice as high as those made in earlier phases of development. [JONES91]

A **pre-award prototype** can be used to determine the offeror's understanding of the requirement, which in turn helps the offeror project a more realistic estimate of system development cost and schedule. While prototyping involves time and resources, ***experience shows that the lead time to a fully operational system is generally less when prototyping is used.*** The prototype allows users and designers to know what is wanted, and having already built a simplified version, the fully-developed system is less expensive and time-consuming. The final product is also more likely to perform as desired with fewer surprises when delivered.

Cautions About Prototypes

Do not mistake a prototype for more than what it is — a shortcut for demonstrating software functions/capabilities and for eliciting user buy-in. Quality control and assurance (testing) and supportability issues (e.g., technical documentation) are seldom addressed, as these activities negate the benefits of the prototype. Deming talked about the *"burn-and-scrape"* method of quality control for toast, comparing it to getting it right the first time. [DEMING82] The requirements for toast are certainly easier to understand than the requirements for most software systems, so some scraping is understandable. However, uncontrolled prototyping can result in an endless, unproductive

sequence of *burn-and-scrape* developments. Remember, prototypes are a *quick-and-dirty* way to evaluate whether the proposed design meets user needs and are generally produced with *throwaway code.* They are also not developed with supportability, readability, and usability in mind, and bypass normal configuration management, interface controls, technical documentation, and supportability requirements. Therefore, you must refrain from expanding a prototype without baselines, interfaces, capacity studies, and thorough documentation. [KINDL92]

NOTE: Refrain from forcing coding standards on prototype development as they can adversely impact the benefits of prototypes.

Another caution about prototypes is they must be well-planned and designed to address significant sources of risks you have thoroughly identified and documented in your **Risk Management Plan**. ***You must make sure every effort has been made to understand the requirements before building any prototype, and then ensure that the prototyping effort is converging on a requirement(s) validation***. To benefit from the prototyping exercise, require that each prototyping effort concludes with the delivery of a **written report** stating what was done, the results, their implications, and the degree to which the prototype met stated objectives. [HUMPHREY90]

When using prototype demonstrations during source selection, the requirement for a development maturity Level 3 or higher *[discussed in Chapter 7, Software Development Maturity]* must be a prerequisite. If supportability (or reliability, portability, interoperability, etc.) are high risk drivers, these capabilities can be included in your functional description of offerors' prototype demonstrations. However be aware, without a sound software engineering process to back it up, source selection prototype demonstrations can be deceptive with false-positive results. An example is the overhaul of Air Force Logistics Command's enormous materiel requirements planning system (a database of approximately 390 gigabytes containing an inventory of one million weapons part items worth $28 billion).

The award of the $210-million contract in 1984 required that each competing firm deliver a working prototype of the proposed system. Although the *compute-off* proved to be a useful method, it illustrates the need to consider the other source selection variables discussed throughout these Guidelines rather than relying on any one

discriminator. Although the company with the best prototype won the contract award, the tactics used to win the 100-yard dash demonstration did not work well in the actual contract performance marathon. Once awarded the contract, the winning company continued to use the same *quick-and-dirty* methods for large-scale production that they used for the prototype demonstration. Thus, they had trouble delivering software with acceptable defect rates and in producing quality documentation. *[There was a happy ending to this example, as the company in question was able to turn deliveries around through process improvement and a commitment to sound software engineering principles.]* [ANTHES90]

HARDWARE REQUIREMENTS

A major concern during software development is how to get the most advanced **computer hardware technology** available. The goal is to get the most *crunch* (computer power) *-for-the-buck*. The question to be answered is how to have the most efficient, advanced equipment possible throughout the system life cycle. Because computer hardware improves at an exponential rate, and user requirements grow and change with technology's leading edge, hardware technology is a major source of change over the system's life. Computer **hardware requirements** definition (e.g., digital systems, digital line replacement units/modules, digital circuit cards, complex digital components) is often a considerable management challenge. The translation of hardware requirements through the design specification down to gate-level schematics requires that designers work across a wide range of **abstraction**. ***The rush to lock into hardware designs before completing essential tradeoffs is often a source of substantial program risk.***

VHSIC/ VHDL

The DoD **Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)**, ANSI/IEEE 1076 (used with commercially available design, analysis, synthesis, and accelerator tools) provides the capability to simulate the hardware design before it is built. The use of **hardware simulation** as part of a disciplined engineering process (particularly for embedded weapon systems computers) can greatly reduce risk and the costs associated with front-end design and analysis of digital systems. VHDL allows simulation and debugging of the hardware design, and definition of inputs/outputs and interfaces before a commitment is made to logic

gates. These tools give designers the ability to employ systems engineering tradeoffs, design simulation, and design iteration, as well as to make design changes and to examine architectural alternatives prior to hardware selection.

Hardware Selection

Ideally, computer hardware selection should be delayed until completing sufficient requirements analysis to predict the processing power and throughput necessary for successful execution of the planned software. Delaying hardware selection might be feasible if contract support is provided through a systems integrator on a cost-reimbursable basis, and flexibility is allowed in timing and selection. More importantly, selecting computer hardware late in the software development process encourages the development of portable software that can be easily migrated among different hardware platforms.

In reality, the recommended hardware is often not only part of the winning contractor's proposal, but an integral part of their cost estimate. If you receive a proposal keyed to a specific hardware set, this can be considered a *weak* proposal. Studies have shown that in major software-intensive systems acquisitions (e.g., weapons systems and C3I systems) the cost of developing software can be as much as 80% of the cost of the hardware and software combined. If an offeror bases their cost estimates on a specific hardware set, they probably do not have a very good understanding of the proposed system.

NOTE: Modifying a system performance requirement is not necessarily a bad thing or a sign of failure. Lessons-learned show that it is often not worth paying 30% more to get the last 5% of originally-specified performance. In software terms, it may be frivolous to spend another million dollars on hardware to reduce terminal response time from 2.1 seconds to the specified 2.0 seconds.

Too often hardware acquisition is conducted separately from the software development process. In this case, the software effort cannot be delayed and completed out of context from its eventual operating environment. Hardware selection must proceed in concert with the software effort which must be completed within hardware environmental constraints (e.g., centralized versus distributed environment, specific database management system, compilers, etc.).

CHAPTER 14 Managing Software Development

NOTE: In recent years, lessons-learned have shown that hardware is procured *too early*. Hardware sits around and waits for the software to be developed, and is effectively obsolete when finally implemented. Another common occurrence is that hardware is often budgeted too early. If hardware is not purchased within the fiscal period for which it was budgeted, the funds are removed from the program.

Factors to consider in hardware selection are **quantitative performance requirements (QPRs)** *[e.g., "throughput," discussed in Chapter 8, Measurement and Metrics]*, especially if C3 requirements are being defined. To properly determine/simulate loading for a QPR measurement, an assessment of how the proposed software/hardware will perform together is essential. **Operating system** upgrades (projected by COTS hardware vendors) must also be considered as they affect future system growth needs. There are three principles to follow in the initial stages of computer hardware selection (which also apply to software architecture design):

- Follow **standards**, either *de facto* or specifically defined,
- Follow an **open systems architecture**, and
- Plan for **evolutionary change** over the software life cycle.

In integrated airborne avionics environments, severe physical and connectivity constraints may exist. Nevertheless, every effort must be made to use **standard computer hardware configurations** with well-understood performance characteristics. Although not similarly affected by physical constraints, some intelligence systems, C2 systems, and MISs must operate with large existing suites of hardware and software. The technical and cost benefits/penalties of compatibility with these pre-existing systems must be assessed. Even when analysis indicates continuing a sole source, proprietary environment is cost-effective, DoD's preference is an **open systems architecture** *[discussed in Chapter 2, DoD Software Acquisition Environment]*.

CAUTION! *"Every vendor with an open mouth claims to have an open system."* [THOMPSON91] **Unless vendors follow industry/government-approved standards** *[discussed in Chapter 5, Ada: The Enabling Technology, and Chapter 2, DoD Software Acquisition Environment]*, **the system is not truly open. On the other hand, the considerable time**

it takes to develop and validate industry standards often leads vendors to use de facto standards. THE POINT IS TO SELECT SYSTEMS THAT ARE "*COMPATIBLE*" AND "*INTERCHANGEABLE*" WITH PRODUCTS FROM A WIDE VARIETY OF VENDORS!

DESIGN

The importance of software design can be stated simply: ***design is where the quality goes in.*** This is the critical activity where your choice of a developer *[based on the source selection criteria discussed in Chapter 13, Contracting for Success, and throughout these Guidelines]* pays off. Skills, experience, ingenuity, and a professional commitment to process improvement and excellence are necessary to ensure your product has ***quality built-in***.

Software design is the pad from which development and maintenance activities are launched. ***Software design is the process through which requirements are mapped to the software architecture.*** Design is also divided into two phases so architecture and requirements allocations are in place before components are detailed. Partitioning the process into two (or more) phases provides the Government with an opportunity to formally review the design as it evolves [e.g., **Preliminary Design Review (PDR)** and **Critical Design Review (CDR)**]. Remedial actions can be taken before the design becomes too detailed. The two-phase process also gives you a chance to subject the high-level design to external review (e.g., by systems and hardware engineering team members). This ensures compatibility with other system software and hardware with which the software must interact. [MARCINIAK90]

The **architectural design** defines the highest-level relationship among major software structural components, representing a holistic view of the system. Refinement of the architecture gives top-level detail, leading to an architectural (preliminary) design representation where software units (CSUs) are identified. Further refinement produces a **detailed design** representation of the software, very close to the final source code. [PRESSMAN92] *[Bottom-up design is this process in reverse.]* Detailed design involves refinements of the architecture (CSUs) leading to algorithmic representations, controls, and data structures for each architectural component. It may be possible to produce poor code from a good design — but seldom is it

possible to produce good code from a poor design. Design is the ***"make-it-or-break-it"*** phase of software development. [GLASS92]

Within the context of architectural and detailed design, a number of activities occur. All the information gathered and analyzed during requirements definition flows into the design activities. The software requirements expressed in the form of information, functional, and behavioral models are synthesized into the design. The design effort produces an architectural design and a detailed design (comprised of a procedural design, a data design, and an interface design). The **procedural design** translates structural components into a procedural representation of the software. The **data design** transforms the domain model (created during requirements definition) into the data structures required to implement the software. **Interface design** not only defines how the software is to interface with other system software and hardware, but with the human-machine interface. Figure 14-3 illustrates how information about the software product, defined during requirements analysis, flows into the design which in turn flows into the coding and testing phases. [PRESSMAN92]

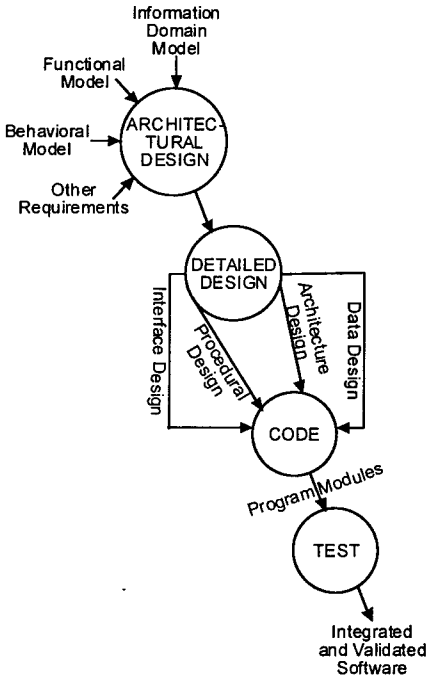


Figure 14-3 Ingredients of Software Design

For software to achieve a high degree of excellence, it must be defect free; i.e., reliable. Adding the dimension of **reliability** to the quality equation (especially for weapon system software) translates into a design requirement for **fault-tolerance**. Software designed to be **fault-tolerant** possesses the following characteristics:

- **Defect confinement**. Software must be designed so that when a defect occurs, it cannot contaminate other parts of the application beyond the module where it occurred.
- **Error detection**. Software must be designed so that it tests itself for, and reacts to, errors when they occur.
- **Error recovery**. Software must be designed so that after a defect is detected, sufficient internal corrective actions are taken to allow successful execution to continue.
- **Design diversity**. Software and its data must be designed so fallback versions are accessible following defect detection and recovery. [GLASS92]

Design Simplicity

When your developer decomposes the solution from the high-level design to lower-levels of detail, it must be kept ***simple***. Whatever design methods employed, the underlying issue is to keep it within human limitations for managing complexity. Since automatic software design has still to evolve as a practical reality, you must apply sound engineering discipline to support your quality design goals. Thus, the **design solution** must be broken down into ***intellectually manageable pieces*** (modules). Ideally, modules should contain no more than **100-200 lines-of-code**. Remember,

Deal with the difficult,
While it is still easy.
Solve large problems
When they are still small.
Preventing large problems
By taking small steps
Is easier than solving them.
By small actions
Great things are accomplished. —Lao Tzu

Throughout the design process, the quality of your developer's evolving design should be assessed though a series of informal in-house walkthroughs, formal technical reviews, and peer inspections *[discussed in Chapter 9]*. To evaluate design quality, ***the criteria for***

a good design must be established in the Software Requirements Specification (SRS). Applying software engineering to the design process encourages the use of fundamental quality design principles, a systematic methodology, and thorough review of the connections between modules and with the external environment.

During design, the **skills** of your software developer are put to the acid test. Quality software must achieve the goals of software engineering by fulfilling the quantifiable principles of well-engineered software *[discussed in Chapter 4, Engineering Software-Intensive Systems]*. To refresh your memory, these include:

- Abstraction and information hiding,
- Modularity and localization, and
- Uniformity, completeness, and confirmability.

Be aware, ***skilled Ada designers require up to 50% more time during design*** than software designers using other languages (which will be recouped during the integration and testing phase). Experienced Ada engineers take the extra upfront time to build the quality you want into the product where it belongs, ***in the design***. Ultimately, the test of whether the software you build is a quality product is **user satisfaction**. But from a software engineering perspective, quality software must possess the measurable quality attributes assigned during requirements analysis that characterize the desired system. *[See Chapter 8, Measurement and Metrics, Table 8-4 for a list of quality attributes.]*

Architectural Design

A good software architecture should reflect **technical certainties** and be independent of variants, such as performance, cost, and the specific hardware selection. It must also address higher-level concepts and **abstractions**. Lower-level details are dealt with during the detailed design phase which defines the particular modules built under the architecture at the software engineering level. By defining only essentials (or certainties), rather than incidentals (or variants), a good architecture provides for the evolution of the system and for the incremental or evolutionary upgrading of components. A sound approach for the software architect is to address (and to commit to) **certain essentials** and to be independent of variable incidentals. ***The hallmark of a good architecture is the extent to which it allows freedom and flexibility for its implementors***.

Architectures must address the relationships among system components (i.e., the **interfaces** between them). **Standardization** of data interfaces, their implementation, access, and communication improves the quality and consistency of data and the overall effectiveness of the system. Data and system interfaces for MIS and C3 systems should be compliant with DISA's **Technical Architecture Framework for Information Management (TAFIM)**. A **standards-based architecture** reflects a managed environment (based on defined standard interfaces) that describes the characteristics of each architectural component. It is depicted through classes of architectural platforms that are, by definition, modular, highly reusable, and inherently flexible. It provides a high degree of **interoperability** in that the architecture is owned by the user — not the vendor. [DMR91]

In building a standards-based architecture you should also make sure your software architecture is built using ***lateral vision*** (i.e., from an agency, command, and user perspective). Once standards-based architectures are built, they must then be integrated into reuse repositories *[discussed in Chapter 9, Reuse]*. A rule of thumb is to use the standards that are out there; e.g., GOSIP, POSIX, SQL, etc.. You should also ensure that your developer makes sound decisions about user interface standards. For MISs, a framework should be picked from the TAFIM. The "*Command Center Store*" of Electronic Systems Command provides a generic architecture and reusable components common to many C2 systems.

A user interface, such as OSF/Motif for XWindows, should also be considered. Client/server roles, a migration strategy application (or model layer), and binary application portability with purchased software are also important factors. NIST, FIPS, and commercial standards should be used, when appropriate. Figure 14-4 illustrates the standards-based architecture planning process. [DMR91]

The baselined architecture and plans for the system's evolution impact your developing application in significant and important ways. Thus, you ***must pay close attention to the architectural design process as it is critical to the success of your program.*** Management considerations include:

- A clear vision of requirements and functionality must be created early,
- Unnecessary complexity must be relentlessly eliminated from systems and software, and
- Careful control of requirements is vital.

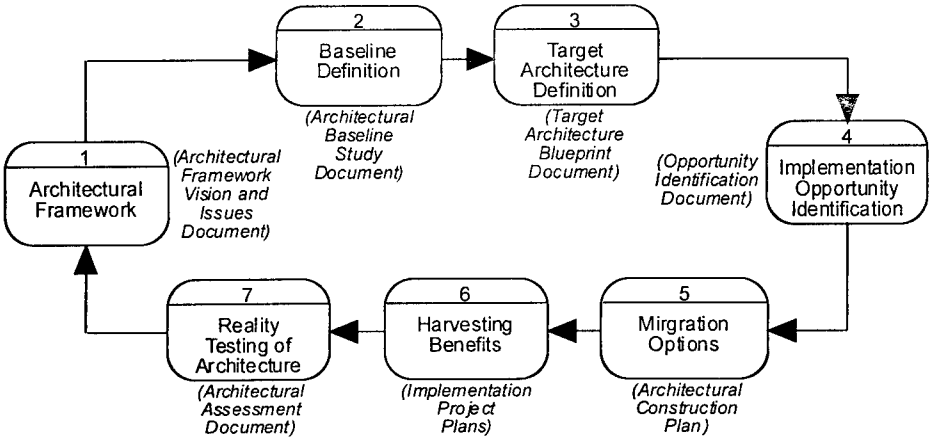


Figure 14-4 Standards-Based Architecture Planning Process [DMR91]

During architectural design, requirements are allocated to a number of components, such as objects or functional requirements. **Derived requirements** are also defined and allocated. Derived requirements may reflect the need to reuse existing components, to design components for reuse, to take advantage of available COTS software, or other factors such as security and safety design constraints. While not originally stated in the requirements specification, derived requirements impact on quality and performance and must be reflected as functions (or objects), mapped from the **SRS** to the architecture.

The product of the architectural design phase is the **software architecture**. The architecture reflects two characteristics of the software solution: (1) the hierarchical structure of procedural components and objects (modules), and (2) the structure of data. It evolves from **partitioning** related elements of the software solution. To achieve openness to COTS, GOTS, and NDI solutions, two forms of **system partitioning** should be achieved. First, systems interfaces, functionality, and data schema must be partitioned within the software architecture such that there are no barriers to the inclusion of the best available technology which requires an awareness of available technology and probable technological progress. Second, the architecture must be partitioned such that those modules that will not change are divorced from the path of those modules slated for evolutionary improvements. In addition, the requirement for an **open systems architecture** *[discussed in Chapter 2, DoD Software Acquisition Environment]* requires that designers possess an early

and knowledgeable awareness of market evolution (and indeed revolution) in the burgeoning software technology arena. [FAA94] The architecture should reflect a tradeoff between needs and currently available technology, whereas interfaces must be designed such that a change within one element of the architecture has minimal impact on other elements.

These related software elements were derived from related elements of the real-world problem domain *[discussed in Chapter 4, Engineering Software-Intensive Systems],* implicitly defined during requirements analysis. The architecture also defines the underlying software structure. It gives a representation of how the software is partitioned (not allocated) into components and their interconnection. It provides a picture of the **flow of data** (including database schema, message definitions, input and output parameters, etc.) and the **flow of control** (including execution sequencing, synchronization relationships, critical timing constraints or events, and error handling). It outlines the **rules for integration** of system components which involves timing and throughput performance attributes and interconnection layers, standards, and protocols. The architecture also distinguishes between hardware structure and software structure, and provides for the future allocation of software components to hardware components. [PAYTON92]

NOTE: See Volume 2, Appendix G, Tab 1, "*The Importance of Architecture in DoD Software*," and Tab 2, "*A New Process for Acquiring Software Architecture.*"

In unprecedented, major software-intensive systems, certain software components often must be custom designed to meet specialized requirements. **CASE tools** should be used to partition and layer the architecture to isolate those functions which are necessarily unique. One of the biggest issues in integrating uniquely developed software with multiple **COTS software** packages is **interoperability** *[COTS is discussed in Chapter 13, Contracting for Success].* A solution to this problem is the introduction of a ***"middle layer"*** of software that isolates the interface between the unique infrastructure and the COTS. The advantage to using such a middle layer is that it **encapsulates** unique protocols so new COTS products can be *plugged-in* as they become available. Mission-unique software must be designed such that the components of the software infrastructure are transparent to the application code. An analogy to the software middle layer can be made to adapting an electrical appliance to a wall

plug. If your radio has a 3-prong plug but your wall circuit only accepts a 2-prong, you can place a 3-prong/2-prong adapter between the two without having to modify either the wall circuit or the radio. *[Refer to Chapter 10, Software Tools, for a description of* ***UNAS****, a tool to assist software architects with a library of pre-existent architectural parts. UNAS also provides the ability to quickly model different architectural solutions in executable Ada code, thus allowing for tradeoffs and demonstrations of alternative approaches.]*

NOTE: See Chapter 2, *DoD Software Acquisition Environment*, for a discussion on open systems architecture requirements.

Addressing Architecture in the RFP

As part of the **RFP**, you should require early delivery and qualification of the software architecture as part of the basic contract. This delivery should be prior to **CDR** of the application software. It is also prudent to require delivery of the proposed architecture in executable Ada code as part of the contractor's proposal for consideration during source selection. **UNAS**, and equivalent middleware, provide the capability for offerors to make such submissions. MITRE suggests that qualification standards for contractor-delivered software include, as a minimum:

- Demonstrated conformance of interfaces to a layered standard *[in addition to the overview layer standard],*
- Demonstrated and validated architectural features,
- Software maintainability,
- Physical reconfigurability,
- Interconnect extensibility,
- Usable performance margin, and
- Software portability. [HOROWITZ91]

MITRE also suggests you require that offerors describe their control process for enforcing open systems architectural standards. Items to look for in offerors' responses include:

- Architecture elements that cannot be changed except by the software architects;
- A standardized method used for communication between applications; and
- Standard tools used to facilitate the integration of application software with the architecture. [HOROWITZ91]

You should further require that the architecture be **configuration managed** and that a prototype model of each architecture be developed. The development (and/or acquisition) of an array of support tools to enhance the integration of applications within the architecture should also be encouraged. [HOROWITZ91] Especially with weapons systems, not all software functions will be solvable with commercially available products. The architecture must, therefore, ensure that unique mission-specific software will interface with commercially available products. The use of isolation layers and automated products for encapsulation are essential. Use of an integrated tool environment, such as the **Rational Environment,**™ *[discussed in Chapter 10, Software Tools]* should be encouraged as it provides the capability for automating the partitioning process, architecture synthesizing, coding of low-level design attributes, and compliance with standards. *[A sample RFP paragraph is found in Volume 2, Appendix M describing the requirement for software development* ***technology scalability*** *(also discussed in Chapter 10, Software Tools).]*

Preliminary Design Review (PDR)

The PDR is a formal government/contractor review conducted for each CSCI to verify the top-level software architecture design is consistent with defined software requirements and suitable for the detailed design. The following topics are covered during the PDR:

- The over software structure to the computer software component (CSC) but not in all cases to the lowest unit level in the software hierarchy *[structure charts are one method for depicting the software architecture]*;
- Functional flow showing how SRS allocated requirements are accomplished.
- Control function descriptions explaining how the executive control of CSCs will be accomplished. Start, recovery, and shutdown features are described for each major function or operating mode.
- Storage allocation, timing, and resource utilization information describing how the loading, response, and scheduling requirements will be met by the selected digital hardware and the software design.
- Software development facilities and tools that will be used for the detailed design, coding, and testing of the software. These facilities and tools include compilers, simulators, data reduction software, diagnostic software, a host computer, and test benches.

- Plans for software testing, with emphasis on integrating CSCs in a phased manner. In particular, top-down, bottom-up, or combination strategies are considered, and an effective strategy for the hierarchical software design selected.
- Human engineering of software-controlled control and display functions. Preliminary versions of user's manuals are reviewed to verify that human factor and training considerations are correctly addressed.

The contractor should answer following questions at the PDR:

- What is the software design structure, the resulting major input/ output flows, and the relationships between CSCs?
- Is the overall software structure consistent with a structured, top-down, object or other design and implementation concept?
- Are all common functions identified and units or subroutines designed to implement these functions?
- Is the interface between CSCs and the operating system or executive clearly defined? Are the methods for invoking each CSC's execution described?
- Has a CSC been designed to satisfy every system requirement?
- Is the traceability relating each CSC to specific software requirements documented?
- Is software being designed in a manner that provides for ease of modification as planned for in the SDP?
- How will the software be integrated with the hardware during full-scale engineering development?
- When will the system and software designs be baselined?
- Are sufficient memory and timing growth capacity being incorporated in the system and software design?
- How will software testing be performed? What levels of testing will be employed? Will an independent analysis and evaluation be accomplished?
- How will testing be used to clearly identify deficiencies as either software or hardware related? How will it be determined if errors/ defects *[defined in Chapter 15, Managing Process Improvement]* are caused by either the hardware or software? How will regression testing be performed?
- How will the software be supported in the field? What hardware and software will be needed for the support base? How will it be procured?

Detailed Design

The **detailed design** is a description of how to logically fulfill allocated requirements. The level of detail in the design must be such that software coding can be accomplished by someone other than the designer. The design of each functional unit (module) is performed based on the software requirements specification and the software test plan. The unit's function, its inputs and outputs, plus any constraints (such as memory size or response time) are defined. The detailed design specifies the logical, static, and dynamic relationships among units. It also describes module and system integration test specifications and procedures.

The software design may be created using **Ada unit specifications** to formally define the interfaces among objects in the solution domain. The advantage of this approach (even at this early stage) is that the system can be initially compiled, with Ada used as a checking mechanism for logical inconsistencies. By using a proper set of SEE design tools, completed unit designs are under configuration management control. In addition, requirements can be tracked and the SEE used to develop external documentation.

As stated above, when designing Ada software and using Ada as the **program design language (PDL)**, ***more resources are needed during the design phase***. Experience shows that experienced Ada designers can improve the quality of the design by highlighting interfaces and by capturing many important design decisions. ***If Ada is used as the PDL, some compilation can begin during design and Ada can be used to detect and correct interface problems early in the life cycle.*** Also, an Ada SEE usually has documentation tools that can traverse a collection of program units and extract important design components, including commented annotations. Therefore, with Ada it is possible to semi-automatically produce design documentation, easing the designer's job. [BOOCH94] A tool (such as **UNAS** or equivalent) should be used that can quickly model alternative design architectures in executable Ada code. Software engineering techniques can then be used to evaluate and make tradeoffs among the different approaches which are finally narrowed down to an optimum solution. As the exploratory process proceeds, the design process becomes more formal. From a quality perspective, the design approach used by your developer must be determined by the nature of the application problem. Your design architecture might be based on either functions, data, objects, or a combination thereof.

Functional Design

For heavily logic-oriented applications (such as real-time systems) where the problem involves algorithms and logic, a **function-oriented approach** is often used. Function-oriented design depicts information (data and control) flow and content, and **partitions** the system into functions and behaviors. From the single highest-level system description, the system is partitioned into functions and subfunctions with ever increasing levels of detail. The value of a function-oriented design is that it provides manageable levels of complexity at each phase of the design process. It also identifies and separates the functions performed at each phase of application execution. This **hierarchical decomposition by function** leaves, however, the question as to what is the most abstract description of the system.

The design focuses on those requirements most likely to change (i.e., around functionality). But, if the specification is poorly written, designers are faced with the problem of having to deal with a top-down design for which they are unable to locate the top. Another problem with hierarchical methods is, as decomposition occurs by defining one level at a time, it can delay the discovery of feasibility problems lurking at lower levels. This can be dealt with by using an iterative process in which low-level problems are addressed with a redesign starting from the top-down. [GLASS92] Another drawback with functional design methods is they have limited software reuse benefits. They can lead to the redundant development of numerous partial versions of the same modules — ***decreasing productivity*** and creating ***configuration management overloads***. [AGRESTI86]

Data-Oriented Design

For heavily data-oriented applications (such as MISs) where the problem involves a database or collection of files, a **data-oriented design** approach is often used. This approach focuses on the structure and flow of information, rather than the functions it performs. A data-oriented design is a clear cut framework: data structure is defined, data flow is identified, and the operations that enable the flow of data are defined. A problem often encountered with a data-oriented approach is a *"structure-clash,"* where the data structures to be processed are not synchronized (e.g., input file is sorted on rank, whereas output file is sorted on time-in-grade). Solutions to the clash problem can be the creation of an intermediate file or the conversion of one structure processor into a subroutine for the other. [GLASS92]

Object-Oriented Design

A variety of **object-oriented (OO) methodologies** and tools are available for software development. Each approach emphasizes different phases and activities of the software life cycle using various terminologies, products, processes, and implementation techniques. The impact of a methodology on the conduct and management of a software development effort can be extensive. Therefore, if you decide to employ an OO approach, you should encourage your developer to investigate and select the OO approach that best fits your specific program needs. [JURIK92] An **object-oriented design** (OOD) method focuses on interconnecting data objects (data items) and on processing operations in a way that modularizes information and processing rather than processing alone. The software design becomes decoupled from the details of the data objects used in the system. These details may be changed many times without any effect on the overall software structure. Instead of being based on functional decomposition or data structure or flow, the system is designed in terms of its component objects, classes of objects, subassemblies, and frameworks of related objects and classes. **Strassmann** explains that component-level software objects can be quickly combined to build new applications. These objects are then candidates for ***reuse on multiple applications*** — lowering development costs, shortening the development process, and improving testing. Because objects are responsible for a specific function, they can be individually upgraded, augmented, or replaced — leaving the rest of the system unchanged. [STRASSMANN93]

Object-oriented technology lets software engineers take a kind of *velcro* (or *rip-and-stick*) approach to software development. The idea is to encase software code into **objects** that reflect real-world entities, such as airplanes, crew chiefs, or engineering change orders. The internal composition of objects is hidden from everyone but the programmer of the object. Once molded into objects, the encapsulated code can be stored in repositories that are network-assessable by other designers. As needed, component-level objects can be quickly grafted with other objects to create new applications. [JENKS93]

Using a familiar graphical user interface, such as windows and icons, the object-oriented approach lets developers visualize and design applications by *pointing-and-clicking* on the objects they wish to use. This approach ***cultivates reuse*** because objects can be used in multiple applications, lowering development costs, speeding up the development process, and improving testing. Because objects are

responsible for a specific function, they can be individually upgraded, augmented, or replaced, leaving the rest of the application unaffected. [STRASSMANN93] ***OOD has the added benefit of allowing users to participate more closely in the development process.*** It is very difficult to describe in writing what a software application is supposed to do, whereas a graphical representation is easy to visualize and manipulate. Objects help all involved in the development process (the systems/software engineers, programmers, and users) to understand what the application should do. [JENKS93]

Object-Oriented Baseball

Object-oriented development differs from traditional functional decomposition methods in that it addresses the identification and definition of **objects**. Firesmith uses the OO simulation of a baseball game to explain the concepts of OOD, as illustrated in Figure 14-5. **Encapsulation** joins **methods** (procedures) and **variables** (data) to create **objects** *[e.g., models of individual players, coaches, managers, umpires, balls, bats, bases, stadiums, and rules]*. Not all objects have the same properties. Some of the objects are **concurrent** with their own thread of control *[players, coaches, managers, umpires]*, whereas other objects are **sequential** *[bats, balls]*. Some objects are **tangible** *[players, balls]*, whereas other objects are **intangible** *[rules, statistics]*. Objects *[players]* also have **attributes**

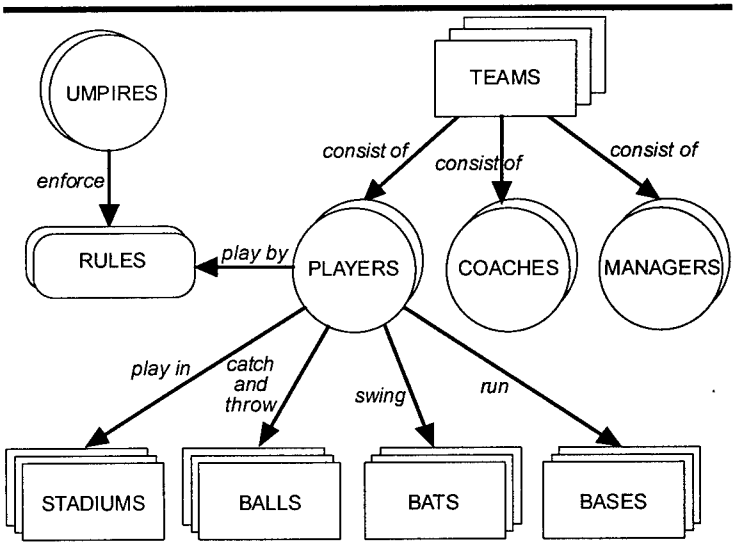


Figure 14-5 Object-Oriented Baseball [FIRESMITH93]

[batting averages, salaries], **operations** *[run bases, pitch, catch, chew tobacco]*, and **exceptions** *[pulled hamstrings]*.

Data and functionality are ***localized*** within the objects rather than being scattered as in functional decomposition methods. This creates a stronger, more powerful form of ***modularity***. Objects *[players]* are black boxes with a *visible* specification that explains both the object's **responsibilities** *[strike 'em out, hit home runs]* and a **hidden body** *[nerves, muscles, bones]*. The attributes and operations are ***encapsulated*** together within the object. How the attributes and operations interact and are implemented is irrelevant (i.e., ***information hiding***) to the outside observer *[fans are not concerned with a player's anatomy, as long as he plays well]*. ***One of the main goals of OOD is to produce objects with well-defined interfaces***. By implementing the software engineering principle of information hiding, program intervals can be hidden within the software leaving the interfaces between modules independent and robust. *[Every time Mussina pitches, the ball always goes over home plate at the same speed, at the same place, and with the same force.]*

Some objects are composed of **aggregates** of other objects *[each team consists of a specific manager and a specific set of players, coaches, and trainers]*, as illustrated in Figure 14-6. In addition, objects usually always fall into **classes** of related objects. When describing an object in general terms, it is typically referred to in terms of **classes-of-objects** *[coaches, players]*, rather than an individual **instance** of the class *[Sparky Anderson, Cal Ripken]*.

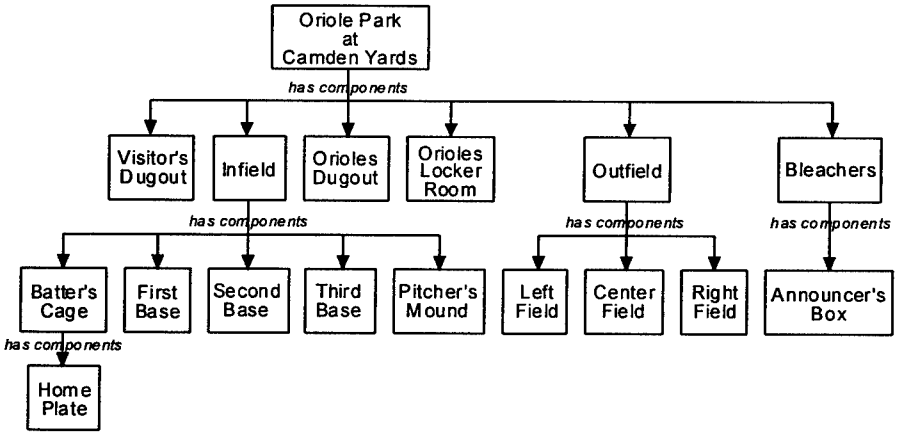


Figure 14-6 Aggregation Hierarchy Example

Understandably, an infinite number of objects can be created from a standard class. Classes usually exist in **hierarchies of superclasses** (superior objects) and **subclasses** (dependent objects) *[pitchers and catchers are subclasses of the superclass baseball players].* Subclasses **inherit** the properties of their superclasses. That is, through *inheritance* methods are passed on to subclasses (called **descendants**) or methods are received from superclasses (called **ancestors**). [STRASSMANN93] Subclasses inherit the attributes, operations, and exceptions of superclasses *[pitchers and catchers inherit the salary ranges (attributes), the required training (operations), and world records (exceptions) of the superclass baseball players].* Figure 14-7 illustrates an example of a classification hierarchy of concurrent classes.

Classification is often difficult because there are different ways to classify the same objects *[baseball players can also be classified as rookies or veterans].* Subclasses have **single inheritance** when they only inherit from a single superclass, whereas they have **multiple inheritance** when they inherit from more than one superclass *[Fernando Valenzuela is a member of the class "pitchers" and a member of the class "athletes," which also includes race horses].* **Dynamic inheritance** occurs when the class an object belongs to changes over time *[as an infant, Reggie Jackson was not a member of the baseball players class].*

Just as umpires can send pitchers, outfielders, catchers, and basemen to the bench with the same message *["Strike three, you're out!"],* **overloading** occurs when the same name is given to different objects, attributes, messages, operations, or exceptions in different scopes. Just as pitchers and batters respond differently when they hear the same phrase, "*Play ball!*" **polymorphism** occurs when different objects respond differently to the same message. *[Ada's strong typing feature helps prevent overloading and enforces polymorphism.]*

Objects and classes are **associated** with each other in various ways. They **collaborate** with one another, often as equal partners, to accomplish their mission *[win the pennant].* Thus, object-oriented software has fewer strict control hierarchies than functionally designed software. For example, the pitcher *[Mike Mussina]* throws the ball to the catcher *[Chris Hoiles]* and Hoiles throws it back to Mussina without each toss having to be directed by the coach *[Johnny Oates].* A **message** is a request by one object to another object to carry out one of its methods. Thus, objects interact by sending messages *[signals]* to each other *[Hoiles signals Mussina to throw a fast ball by*

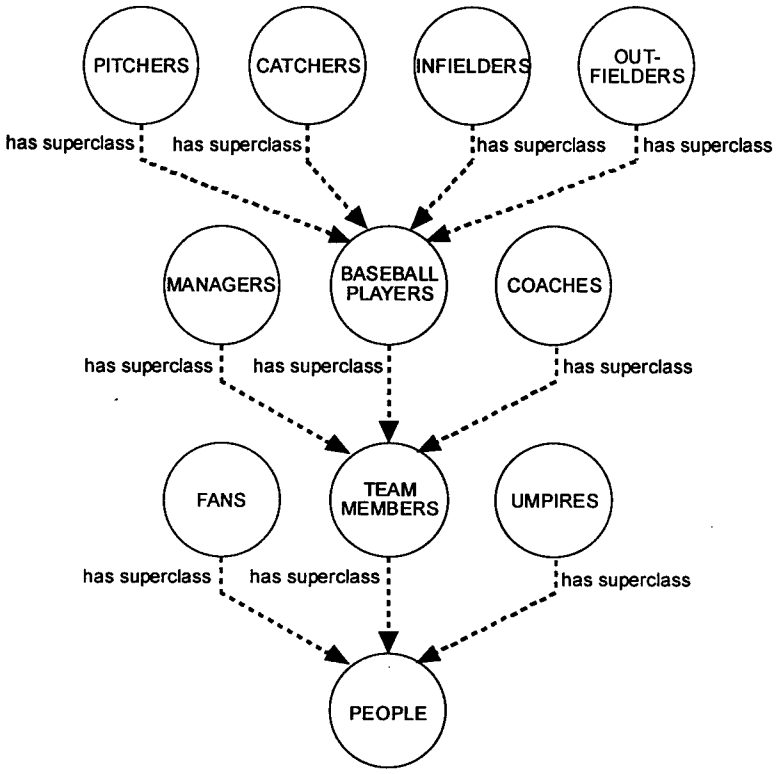


Figure 14-7 Classification Hierarchy Example [FIRESMITH93]

rubbing his nose; the umpire yells, "You're out!" to Cecil Fielder, the opposing batter]. Figure 14-8 shows messages sent from two concurrent objects (circles) to two sequential objects (boxes).

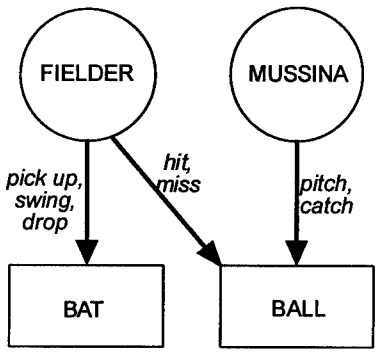


Figure 14-8 Message Passing Example

Because a great number of objects and classes exist in any complex application, software developers must be able to organize them into manageable collections. Objects and classes can be grouped into **subassemblies** which in turn can be grouped into **assemblies** *[players can be organized into teams and teams organized into leagues]*. Application **frameworks** are reusable designs that occur over and over again in the same solution domain, just as the basic structure of baseball teams is repeated over again in the various leagues around the world. [FIRESMITH93]

Problem Domains and Solution Domains

Object-oriented development pioneer, **Grady Booch**, explains how OOD methodology facilitates developers in solving real-world problems through the creation of complex software solutions. The concept of problem domains and solution domains is illustrated in Figure 14-9. The **problem domain** has a set of real-world objects,

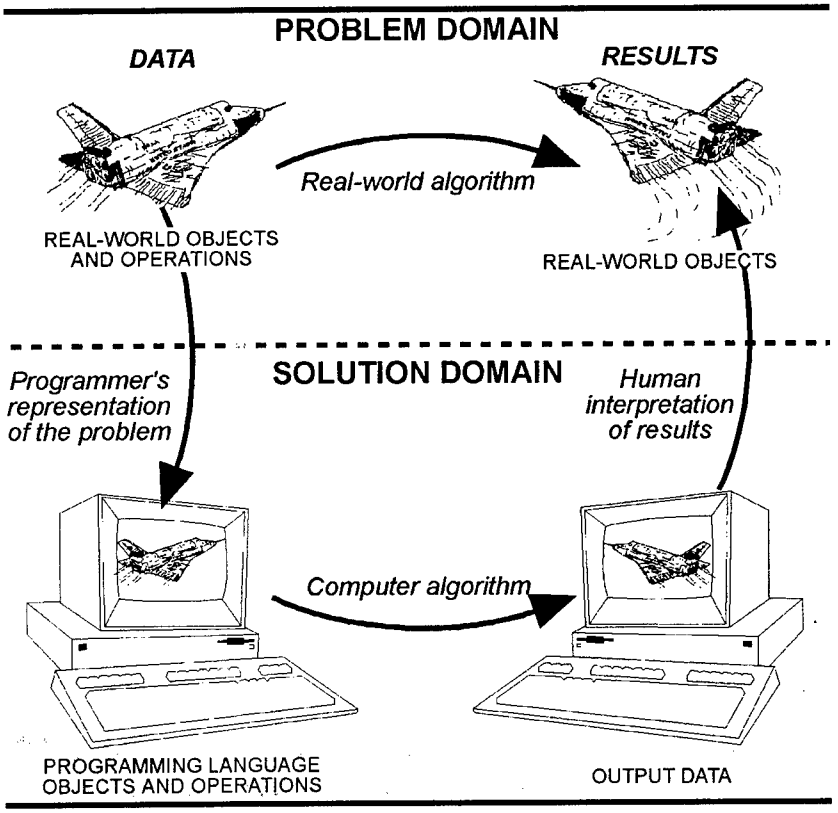


Figure 14-9 Problem Domain/Solution Domain Analytical Process

each with its own set of appropriate operations. These objects can be as simple as a baseball bat or as complicated as the **Space Shuttle**. Also in the problem domain are real-world algorithms that operate on the objects, resulting in transformed objects. For example, a real-world result may be a course change for the Space Shuttle. When developing software, either the real-world problem is modeled entirely in software, or for example in embedded software, real-world objects are transformed into software and hardware to produce real-world results. ***No matter how the solution is implemented, it must parallel the problem domain.*** Programming languages provide the means for abstracting objects in the problem domain by

Figure 14-10 Space Shuttle Defect Detection/Removal/ Prevention Is Critical

implementing them into software. Algorithms, which physically map some real-world action (such as the movement of a control surface), are then applied to the software object to transform it. The closer the **solution domain** maps your understanding of the problem domain, the closer you get to achieving the goals of modifiability, reliability, efficiency, and understandability.

OOD differs fundamentally from traditional development, where the primary criterion for decomposition is that each software module represents a major step in the overall process. With OOD, each system module stands for an object or class of objects in the problem domain. [BOOCH94] Of course, you will not always have perfect knowledge of the problem domain; instead, it may be an iterative discovery process. As the design of the solution progresses into greater states of decomposition, it is likely new aspects of the problem will be uncovered that were not initially recognized. However, if the solution maps directly to the problem, any new understanding of the problem domain will not radically affect the architecture of the solution. With an object-oriented approach, developers are able to limit the scope of change to only those modules in the solution domain that represent changing objects in the problem domain. *[The Space Shuttle mission will always be fulfilled by a space vehicle (constant); how that vehicle is propelled (variable) may change as technology advances.]*

The OOD method supports the software engineering principles of abstraction and information hiding, since the basis of this approach is the mapping of a direct model of reality into the solution domain. This strategy also provides a method for decomposing a software system into modules where design decisions can be localized to match our view of the real world. It provides a uniform means of notation for selecting those objects and operations that are part of the design. With Ada as the design language, the details of operations can be physically hidden, as well as, the representation of objects. Figure 14-11 (below) summarizes how the OOD approach reduces risks and lowers costs. [SA92]

Critical Design Review (CDR)

The purpose of CDR is to verify that the detailed software design is complete, correct, internally consistent, satisfies all requirements, and is a suitable basis for coding. The CDR follows the Detailed Design phase, and the successful completion of CDR marks the completion of the Detailed Design phase. The CDR is performed to establish the integrity of a computer program design before coding

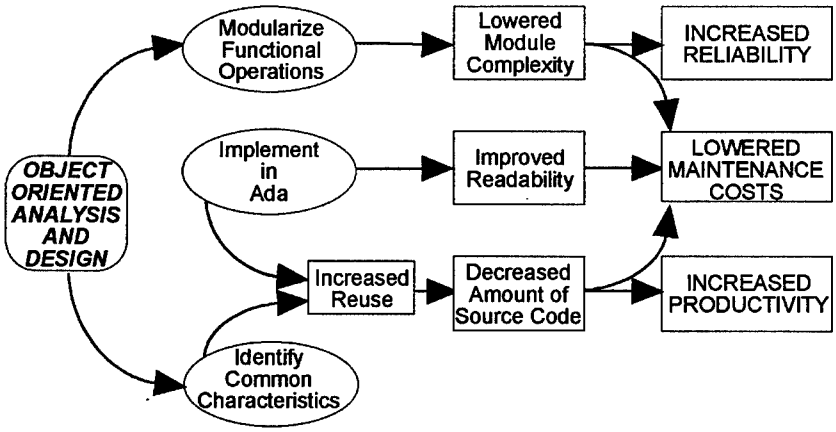


Figure 14-11 Benefits of Using an OOD Approach with Ada

and testing begins. When a given software system is so complex that a large number of software modules will be produced, the CDR may be accomplished in increments during the development process corresponding to periods during which different software units reach their maturity. For less complex products, the entire review may be accomplished at a single meeting. The primary product of CDR is the formal review of specific software documentation, which will be approved and released for use in coding and testing. CDR covers the following topics:

- Description of how the top-level design, presented at the PDR, has been refined and elaborated upon to include the software architecture down to the lowest-level units.
- The assignment of CSCI requirements to specific lower-level CSCs and units.
- The detailed design characteristics of the CSCs. These detailed descriptions shall include data definitions, control flow, timing, sizing, and storage allocation. Where the number of units is large and the time for the CDR limited, the description concentrates on those units performing the most critical functions.
- Detailed characteristics of all interfaces, including those between CSUs, CSCs, and CSCIs.
- Detailed characteristics of all databases, including file and record format and content, access methods, loading and recovery procedures, and timing and sizing.
- Human engineering considerations.

- Life cycle support considerations that include a description of the software tools and facilities used during development that will be required for software maintenance.

The contractor should answer the following questions at CDR:

- Are each unit's inputs/outputs clearly defined? Are the units, size, frequency, and type of each input/output parameter stated?
- Is the processing for each unit defined in sufficient detail, via flow charts, programming design language (PDL), structured flow charts, or other design language so that the unit can be coded by someone other than the original designer of the unit?
- What simulations, models, or analyses have been performed to verify that the design presented satisfies system and software requirements?
- Has machine dependency been minimized (e.g., not overly dependent on word size, peripherals, or storage characteristics)? Have machine dependent items been segregated into independent units?
- Has the database been designed and documented? Has it been symbolically defined and referenced (e.g., was a central data definition used)?
- Have the software overall timing and sizing constraints been subdivided into timing and sizing constraints for individual units? Are the required timing and sizing constraints still met?
- Have all support tools specified for coding and debugging (i.e., pre- and post-processor) been produced? If not, are they scheduled early enough to meet the needs of the development schedule?
- Are the software test procedures sufficiently complete and specific so that the test can be conducted by someone else?
- Do the test procedures include input data at the limits of required program capability? Do test procedures contain input that will cause the maximum permitted values and quantities of output?
- Do test procedures exercise representative examples of all possible combinations of both legal and illegal input conditions?
- Are there any potential software errors that cannot be detected by the test runs in accordance with the test procedures? If so, why? What will be done to make certain the software does not have those errors?
- How will detected errors be documented? How will corrective actions be recorded and verified?
- What progress has been made in developing or acquiring the simulations and test data needed for testing? Will they be available

to support these testing efforts? How will they be controlled during the test effort?

TESTING

Testing has been the most labor-intensive activity performed during software development. As illustrated on Table 14-1, testing often requires more effort than the combined total for requirements analysis and design by as much as 15%. It has also been a significant source of risk, often not recognized until too late into cost and schedule overruns. There are two basic reasons why testing is risky. First, testing traditionally occurs so late in software development that defects are costly and time consuming to locate and correct. Second, test procedures are *ad hoc*, not defined and documented, and thus, not repeatable with any consistency across development programs. We enter testing without a clear idea of what and how it is to be accomplished. Testing can be a major source of wheel spinning which can lead from one blind alley to another.

	LIFETIME COST (%)	DEVELOPMENT (%)
Development	20%	
Analysis and design		35%
Coding		15%
Testing		50%
Operations	Insignificant	
Maintenance	80%	
NOTE: Development productivity is 10-15 lines-of-code per person-day. Maintenance estimates vary from 40% to 80%.		

Table 14-1 Software Development Life Cycle Cost [BOEHM76] [BROOKS75]

Historically, software testing has been a process that ***checks software execution*** against requirements agreed upon in the SRS. The goal of software testing was to demonstrate correctness and quality. Today, we know this definition of testing is imprecise. Testing cannot produce quality software — nor can it verify correctness. Testing can only confirm the presence (as opposed to the absence) of software **defects**. The testing of source code alone cannot ensure quality software, because testing only finds faults. It cannot demonstrate that faults do not exist. Therefore, correcting software defects is a fix, not a solution. ***Software defects are usually symptoms of more fundamental problems in the development process***. Development process problems might be the failure to follow standard procedures, the

misunderstanding of a critical process step, or a lack of adequate training.

Thus, the role of software testing has evolved into an integrated set of ***software quality activities*** covering the entire life cycle. Software tests apply to all software artifacts. To engineer quality into software, you must inspect, test, and remove errors and defects from requirements, design, documentation, code, test plans, and tests. You must institute an effective **defect prevention program** that engages in accurate defect detection and analysis to determine how and why they are inserted. [KINDL92] Remember, "*Error is discipline through which we advance.*" [CHANNING92] Although testing cannot prevent defects, ***it is the most important activity for generating the defect data necessary for process improvement.***

Developmental testing must not interfere with, nor stand apart from, daily development activities; it must be embedded within your development process. Furthermore, given the uniqueness of each DoD software development program, the embedded testing methodologies you apply must be customized to your environment. If testing standards are instituted and the testing process is properly planned, the time and effort required for testing can be significantly reduced. [MOSLEY93]

Testing Objectives

Because testing is not limited to the testing phase, but spans the entire software development, your developer's Test Plan must state general objectives for the overall testing process and specific objectives for each development phase. The primary objective should be to assess whether the system meets user needs. Other objectives depend on the software domain and the environment in which the system will operate. Testing objectives also focus on verifying the accomplishment of quality attributes, as discussed in Chapter 8, *Measurement and Metrics*. The bottom line with testing is ***test early***, ***test often***, and ***use a combination*** of testing strategies and techniques. Also, ***automate every testing activity*** economically and technically feasible.

NOTE: See Chapter 10, *Software Tools*, for a discussion on automating the testing process.

Defect Detection and Removal

Defect detection and **removal** is the most basic testing objective and the one aspect of quality that can be measured in a tangible and convincing way. Defects (and their removal) can be measured with great precision, and their measurement is one of the fundamental parameters to include in every testing and measurement program. Programs performing well in defect removal normally perform well in other aspects of quality, such as requirements conformance and user satisfaction. Conversely, programs with inadequate defect removal are seldom successful in achieving other quality goals. [JONES91] Therefore, you must make sure your contractor measures and documents defects throughout the life cycle.

It is important to understand that ***"errors"*** relate to early phases of development: requirements definition and design specification. An error in requirements or design will cause the insertion of one or more "***defects***" in the code. However, a defect may not be visible during code execution — neither during testing nor operation. If a defect is executed, it may result in a tangible fault, or it may not. Programmers ***debug*** code to correct defects by testing for tangible failures. But the lack of failures cannot guarantee the absence of defects. Even if the defect executes, it may not be visible as output. Furthermore, defect correction does not necessarily imply that the error (source of the defect) causing the defect has been corrected.

There are three broad classifications of defects, named after the development phase where they are found: unit/component defects, integration defects, and system defects. **Unit defects** are the easiest to find and remove. When system testing and a test is failed, you cannot tell if the failure is caused by a unit, integration, or system defect. It is only after the failure is resolved that we know from where it came. As discussed above, system testing is more expensive than unit testing and any unit defect remaining during system testing translates into costly scrap and rework *[discussed in Chapter 8, Measurement and Metrics]*. **Integration defects** are more difficult to detect and prevent because they occur from interaction among otherwise correct components. Component interactions are ***combinatoric*** — i.e., they grow as n^2 (the square of the number of components integrated) or worse (e.g., $n!$ — that number factorial). An integration testing objective is to assure that few, if any, harmful component interaction defects remain before going to system testing. During system testing, we have the added complexity of multitasking, i.e., the order in which things happen can no longer be predicted with

certainty. This uncertainty and the issue of timing is rich soil for ever more complex **system defects**. [BESIER95]

NOTE: See Volume 2, Appendix O, *Additional Volume 1 Addenda*, Chapter 4 Addendum B, "*Software Reliability: A New Software OT&E Methodology.*"

You might ask, if the defect cannot be detected and does not show itself as output, why bother removing it? With mission and safety critical software operating under maximum stressed conditions, the chances of a latent defect-related software failure often increases beyond acceptable limits. This dichotomy amplifies the need to detect, remove, and ultimately prevent the causes of errors before they become illusive software defects. Latent, undetected defects have the tendency to crop up when the software is stressed beyond the scope of its developmental testing. It is at these times, when the software is strained to its maximum performance, that defects are the most costly or even fatal. [KINDL92]

NOTE: See Chapter 15, *Managing Process Improvement*, for further discussion of defect causal analysis, defect removal efficiency, and defect prevention.

The number of errors (unintentionally injected into software by requirements analysts and designers) and defects (injected by programmers while interpreting designs) can be quite large. For complex software systems they can number in the tens-of-thousands. [PUTNAM92] Jones has noted defects for varying sizes of systems to range from 50 to 95 defects per KLOC. [JONES86] Most of these, however, are removed before delivery by the self-checking of analysts and programmers, by design reviews, peer inspections, walkthroughs, and module and integration testing. Jones estimates the pre-delivery **defect removal rate** using these techniques to be at about 85%.

For systems where failure to remove defects before delivery can have catastrophic consequences in terms of failed missions or the loss of human life, defect removal techniques must be decidedly intense and aggressive. For instance, because the lives of astronauts depend implicitly on the reliability of **Space Shuttle** software, the software defect removal process employed on this program has had a near perfect record. Of the 500,000 lines-of-code for each of the six shuttles delivered before the Challenger, there was a **zero-defect**

rate of the mission-unique data tailored for each shuttle mission, and the source code software had 0.11 defects per KLOC. [KOLKHORST88]

NOTE: This is not to imply the Challenger disaster was caused by software defects. It was merely the cutoff point for the report upon which this example is based.

These impressive figures reflect a formal software engineering process that concentrates on learning from mistakes. Finding and correcting mistakes must be a team effort where no individual is held responsible or singled out. Management must treat finding defects as a positive activity left to the team's discretion (management and the user are not included). Figure 14-12 illustrates the steps performed for every software defect found on the Space Shuttle program, regardless of significance. Process improvement is relentlessly achieved by performing feedback during steps 2 and 3. Much credit for this achievement is attributable to peer inspection techniques *[discussed below]*, pioneered by IBM-Houston.

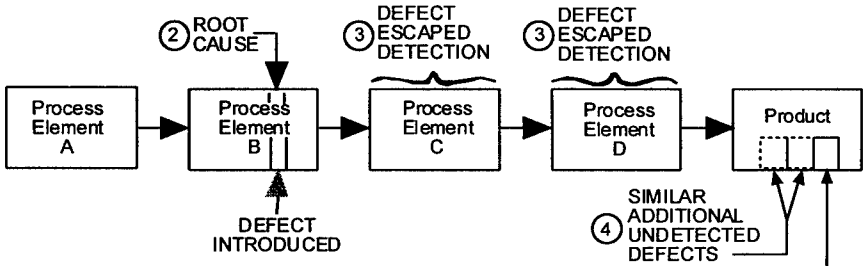


Figure 14-12 Space Shuttle Defect Removal Process Improvement [KELLER93]

NOTE: See Chapter 15, *Managing Process Improvement*, for a discussion on the Cleanroom statistical software testing and verification process.

Defect Removal Strategies

Given the magnitude of errors associated with requirements and design, it is obvious that these huge sources of errors must be included in your **quality control/assurance** strategies. PAT teams, demonstrations, prototypes, and peer inspections are all necessary to control front-end sources of errors. Testing and peer inspections are also necessary for discovering inserted defects. It is important to recognize the upfront costs of inspections and testing, as well as the expected downstream cost, quality, and schedule benefits. [BRYKCZYNSKI93] Incentives should be provided to contractors who demonstrate savings through the inspection and defect removal data they produce and validate.

Finding and removing defects is the most expensive activity in which the software industry invests. However, organizations who engage in quality control and defect prevention have an enormous competitive advantage over those who do not. Given the low average efficiencies of most defect removal methods, it is imperative that your developer use a variety of removal techniques to achieve a high cumulative removal efficiency. Special attention must be given to the defects accidentally introduced as the by-products of fixing previous defects. ***The total quantity of bad fixes averages about 5% to 10%, which directly relates to the complexity of the product being repaired.*** Leading commercial and DoD software developers, for example, may include as many as 20 to 25 different defect removal activities. Serious quality control requires a combination of many techniques each aimed at a class of defects for which its efficiency is the highest. The bottom line in choosing your defect prevention and removal strategy is to choose the combination of methods which will achieve the highest overall efficiency and quality gains for the lowest total life cycle cost. [JONES91]

Developer Testing

Testing is usually divided into two activities — unit testing and systems testing. **Unit testing** is often accomplished in an incremental design/code/test fashion, where more and more of the completed system is progressively tested during each increment. Test cases are selected to force the occurrence of defects. The results of unit tests are then analyzed to see if any defects have occurred, and a **debugging process** is performed to remove them. A description of the type, cause, and correction of defects is then placed in a database for future process

improvement analysis. The purpose of unit testing is to remove all defects from the component under test. The easiest way to accomplish this is to ***begin as early as possible*** with **requirements testing** of the component. **Component requirements** are easily tested as they represent but a small subset of the requirements for the whole software product. Structure-driven, statistic-driven, and risk-driven testing are also performed during unit testing. [GLASS92] There are two basic types of testing performed at the unit and system level: structural testing (also called glass-box or white-box testing) and behavioral testing (also called functional or black-box testing).

Structural testing, or *testing-in-the-small*, ideally involves exhaustively execution of all paths of control flow in a module or system. In reality, exhaustive path testing is impossible because the number of potential paths can be infinite. For example, consider an Ada paragraph containing 10 alternate pathways. This module has 10 billion potential operating states because for each iteration, up to a maximum of 10, there are 10 potential paths that can be executed (10 to the 10^{th} power). If you could generate and execute one test case every minute, it would take over 19,000 years to test just this one module. Also, path testing cannot detect missing paths and cannot detect data sensitivity defects. Thus, structural test case design must be based on random and/or selective testing of control flow. Structural testing techniques include:

- Statement coverage,
- Decision coverage,
- Condition coverage,
- Decision/condition coverage,
- Multiple decision/condition coverage,
- Independent path coverage, and
- Structured tableau. [MOSLEY93]

Behavioral testing, or *testing-in-the-large*, focuses on requirements. For example, testing consists of testing all features mentioned in the specification. Behavioral testing be performed, in theory but not in practice, with total ignorance of how the object under test is constructed. It is not concerned with the internal structure of behavior of the module or system, but only with the instances when the program or module does not behave as indicated in its functional specifications. In contrast with exhaustive path testing, behavioral testing focuses on exhaustive input testing, which is also an impossible task. The number of possible valid inputs approaches infinity, as does the number of all possible invalid inputs. Thus, behavioral test case

design must be based on random and/or selective testing of inputs. Behavioral testing techniques include:

- Equivalence partitioning,
- Boundary analysis,
- Cause effect graphing,
- Structured tableau, and
- Error guessing. [MOSLEY93]

Neither testing approach alone is enough. Behavioral testing should be used throughout development, while structured methods are best used later in the process. Both methods are complementary; however, some redundancy of test case design exists between certain techniques within the two approaches. The tester should select and use a combination that maximizes yield and minimizes redundancy. Again, automated tools that build test cases are a sound investment.

Unit Testing

A **unit** is a component. A component is an aggregate of one or more components that can be tested as an aggregate, such as subroutines, functions, macros, the application and the subroutines it calls, communicating routines, or an entire software system. In unit testing is usually performed by the programmer who created the unit.

The **Cargo Movement Operations System (CMOS)** is a sound example of why ***early unit testing pays off***. The CMOS SPO, HQ AFMC's **Air Force Logistics Information File (AFLIF)**, Federal Express, and United Parcel Service participated in a software prototype demonstration of transportation data exchange using the EDI. To meet their challenge, a new testing methodology was incorporated within the CMOS SPO's software development process.

During requirements definition, mobility planners from HQ USAF, Air Combat Command, **Standard Systems Center (SSC)**, and base-levels redefined and refined support requirements and produced the first iteration of an integrated prototype concept of operations. During design, specifications were developed (through a series of technical interchange meetings and system status reviews) which were easily included in the baseline documentation. During the code and test phase, the contractor coded and tested the new software. End users validated the user interface, and CMOS SPO functional analysts tested and validated each of the contractor's modules as they were

completed. This process was labeled *"Unit Test 3 (UT3)."* UT3 represents a significant departure from the normal software development life cycle where software is fully developed and delivered to the Government before **functional certification testing (FCT)** is performed. By using the UT3 approach, problems identified by the SPO were immediately corrected by the contractor.

Qualification testing was accomplished in two phases. The first 30-day test phase (performed in an in-house, integrated laboratory testing environment) was accomplished using actual Seymour-Johnson AFB deployment data and scenarios. SPO personnel and operators participated. Corrections to software problems and operator-requested changes were accomplished quickly, and then recycled into the integrated testing environment. The second phase (conducted at Seymour-Johnson) included paperwork and live exercises which used actual Seymour-Johnson AFB deployment data and scenarios. Software problems and operator-requested changes were corrected at the contractor's facility and sent via modem back to the base in a matter of days.

The new CMOS unit testing process was expected to deliver a better product, decrease development time by 30 days with fewer problems, and incorporate enhancements resulting from earlier user involvement. To prove these assumptions, some modules were omitted from the UT3 process. Table 14-2 summarizes the software deficiencies found during integration testing of the modules that did and did not go through the UT3 process. Changing the SPO's functional testing to perform it early in the design and code phases (rather than after formal delivery) produced the following successes:

SOFTWARE PROBLEM REPORT (SPR) CATEGORY	% of TOTAL not UT3'd	NO UT3	UT3'd	TOTAL SPRs
Critical	100.00%	3	0	3
Significant	60.00%	3	2	5
Minor	65.00%	13	7	20
Enhancement	56.45%	35	27	62

Table 14-2 CMOS Integration Test Results of UT3/ Non-UT3 Modules

- The design and test schedule was decreased by 30 days;
- No critical errors were discovered during qualification testing in modules that underwent the UT3 process;
- The number of significant and minor errors was reduced; and
- Hands-on training for SPO personnel started before formal software delivery.

This success story led to other changes in the CMOS software development process. FCT will now be replaced by UT3 and a new approach to writing test plans is also underway. Before further UT3 begins, a test description shell will be written based on operational scenarios. The detail of the test descriptions will then be added as the scenario undergoes UT3 testing. This leads to accurate test descriptions while decreasing the overall time for software coding and testing. [HEITKAMP94]

Cautions About Unit Testing

Not all unit testing has been as successful as the CMOS effort. One problem with contractor unit testing is that often the tester is biased. Most unit tests (called white box testing) are designed by the same programmers who produce the code, as they are believed to be the only ones who understand it. This proves unsuccessful because, unfortunately, the people who create defects are the least likely to recognize them. Another problem is coverage. The ideal unit test data set is one that exhaustively executes every path of control flow in a module. In reality, this is impractical and nearly impossible because the number of possible paths is often infinitely large. [HUMPHREY90]

NOTE: In the book, Microsoft Secrets, Cusumano and Selby say that Microsoft produces a daily software build each night, and makes the developer use it the next day. [CUSUMANO95] **This early testing pays off in that problems are found earlier. This is also one of the techniques inherent in Cleanroom engineering** [discussed in Chapter 15, *Managing Process Improvement*].

Integration Testing

While unit testing is performed by programmers on the modules they develop, **integration testing** is performed to determine how the individual modules making up a subsystem component perform together as an integrated unit. With large software developments,

integration testing often involves the software of many developers where individually developed modules are combined into various software subsystems and tested as integrated units. As a manager, you must be aware of the political problems associated with integration testing of multiple vendor products. Often, when a defect occurs on the interface between two supposedly pretested and correct components, neither developer wishes to take the blame for the defect and finger pointing occurs. Each developer believes they have a perfect module and that the defect must have been caused by — and thus must be fixed by — the other guy. This situation takes a mix of tact and diplomacy on the integration test manager's part to resolve these problems and get the defects corrected. [GLASS92] Figure 14-13 illustrates how software test stations are used to support all aspects of software integration testing on the **F-16** avionics program.

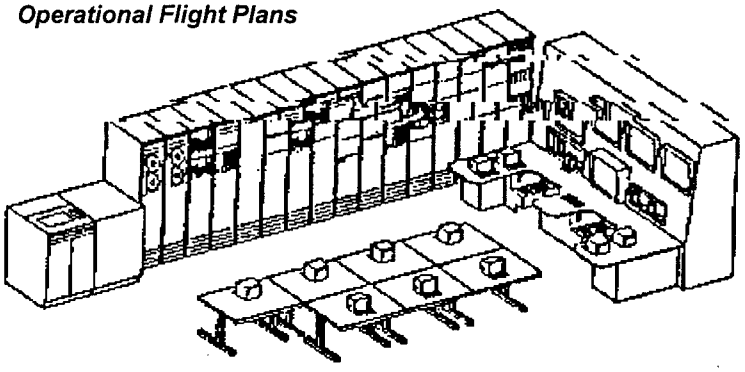
System Testing

System testing (conducted by the systems developer) usually begins after integration testing is successfully completed. Some redesign and tweaking of both the hardware and software is performed to achieve maximum levels of performance and to iron out bugs. System testing is very much like integration testing where components are integrated into whole parts, but not necessarily whole *software* parts. As with integration testing, the system tester tries to invoke defects while the individual component developers are responsible for their repair. The system tester is also responsible for resolving any political problems that arise. *[Remember, it is essential to perform adequate end-to-end testing prior to signing-off on standard form DD-250s for software.]* Figure 14-14 (below) illustrates the systems integration and testing process for F-16 avionics.

Government Testing

Operational system testing of major Air Force software-intensive systems is conducted by an interdisciplinary, independent testing agency. The **Air Force Operational Test and Evaluation Center (AFOTEC)** is a separately operated agency that reports directly to the Chief of Staff of the Air Force. The Center is comprised of a headquarters at Kirtland AFB, New Mexico, detachments at operational locations, and AFOTEC test teams at designated test sites. AFOTEC plans, directs, controls, independently evaluates, and reports on the operational test and evaluation (OT&E) of all major Air Force weapon systems, weapon system support systems, C2, and MISs. It

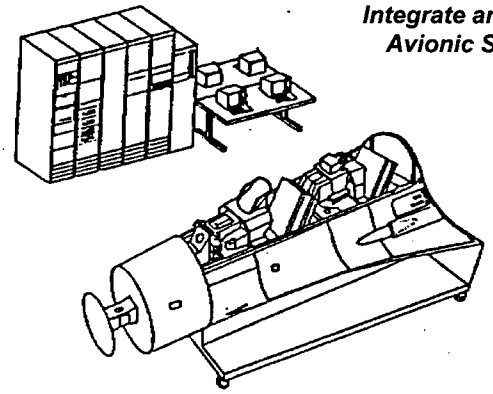
Verify and Qualify Operational Flight Plans



F-16 Software Test Stations Support All Aspects of Software Testing

Engineering Release Test:	Formal Qualification Test:
• Verifies preparation for integration	• Verifies preparation for production
• Executed for each development release	• Executed for each development release
• Isolates software development errors	• Isolates noncompliance with requirements

Integrate and Qualify Avionic Systems



F-16 Interim Test Station Verifies Avionic Suite Capabilities

- Electrical Interface
- Data bus duty cycle
- System design
- Pilot/vehicle interface
- System fault detection
- Mode switchology and symbology
- Hardware failure effects and recovery capabilities
- Verifies flight test preparation

Figure 14-13 F-16 Avionics System Integration Testing

Avionics Flight Test

Ground Test Verifies Compatibility of New/Modified Avionics:
- Safety of flight
- Electromagnetic compatibility
- Aircraft wiring
- Hardware functionality

Flight Test Verifies Avionics Performance in Flight Environment:
- Proper operation, symbology, and switchology for all system modes and functions
- Weapon delivery functions and accuracies
- Pilot-vehicle interface operational usability

Figure 14-14 F-16 Avionics Integration and System Testing

supports system development and production decisions by providing operational assessments and initial OT&E to determine operational effectiveness (how well the system performs) and suitability (including reliability, maintainability, and supportability). Table 14-3 lists the AFOTEC publications you should consult for guidance on software OT&E procedures.

AFOTEC Testing Objectives

AFOTEC cites five objectives in testing system software.

- Usability,
- Effectiveness,
- Software maturity,
- Reliability *[discussed in Chapter 4, Engineering Software-Intensive Systems],*
  - Safety *[discussed in Chapter 4, Engineering Software-Intensive Systems]*, and
- Supportability *[discussed in Chapter 11, Software Support].*

PAMPHLET	TITLE
AFOTECP 99-102, Volume 1	Management of Software Operational Test and Evaluation
AFOTECP 99-102, Volume 2	Software Support Life Cycle Process Evaluation Guide
AFOTECP 99-102, Volume 3	Software Maintainability Evaluator's Guide
AFOTECP 99-102, Volume 4	Software Usability Evaluator's Guide
AFOTECP 99-102, Volume 5	Software Support Resources Evaluation Guide
AFOTECP 99-102, Volume 6	Software Maturity Evaluation Guide
AFOTECP 99-102, Volume 7	Software Reliability Evaluation Guide
AFOTECP 99-102, Volume 8	Software Operational Assessment (SOA) Guide

Table 14-3 AFOTEC Software OT&E Pamphlets

Usability

Useability evaluations concentrate on the operator's interaction with a software-intensive system. Observation of test events should reveal strengths and limitations of the system's operator-machine interface and its supporting software. A usability questionnaire is used to assess the usability characteristics of confirmability, controllability, workload suitability, descriptiveness, consistency, and simplicity. *[See AFOTECP 99-102, Volume 4, Software Usability Evaluator's Guide.]*

Effectiveness

Effectiveness evaluations concentrate on ensuring all critical software is exercised in operationally representative scenarios. Software effectiveness is determined by (and dependent on) system effectiveness.

Software Maturity

Software maturity *[as opposed to software development maturity discussed in Chapter 7]* is a measure of the software's evolution towards satisfying all documented user requirements, as illustrated in Figure 14-15. *[Refer to AFOTECP 99-102, Volume 6, Software Maturity Evaluation Guide.]* The main AFOTEC indicator of software maturity is the trend in accumulated software changes to correct deficiencies, provide modifications, and accommodate hardware changes. The software maturity test objective considers software fault trends, severity, and test completeness while taking into account planned software modifications.

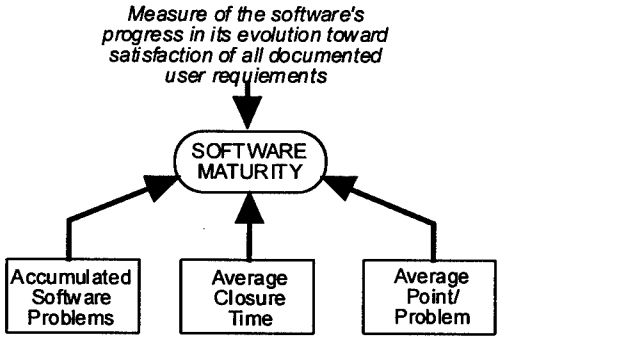


Figure 14-15 OT&E Process for Software Maturity

Software maturity uses a **severity point system** to track unique problems. A weighted value is assigned based on the severity of the failure as defined in the **Data Management Plan**. Software faults of higher severity are assigned a higher value than those of less severity. As the test progresses and new fault data are collected, they are plotted against a time line. Ideally, the slope of the curve should decrease with time. This maturity assessment method is illustrated in Figure 14-16.

AFOTEC Software Evaluation Tools

The AFOTEC software evaluation tools *[AFOTECP 99-102, Volumes 1-8]* should be used throughout the acquisition and development phases of major systems software. They are based on COTS software metrics, ensure credible evaluations, and help to reduce life cycle costs and schedule. Software evaluation approaches differ among

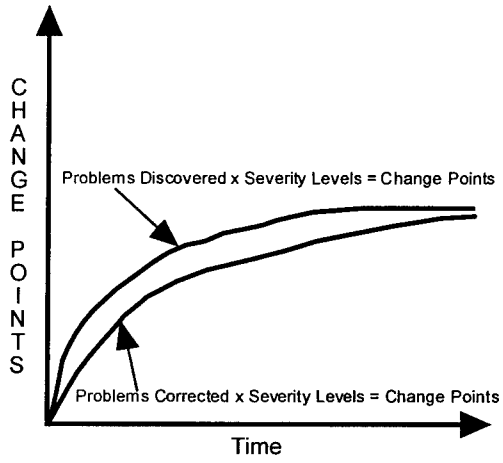


Figure 14-16 Software Maturity

programs; however, the AFOTEC mission is to evaluate software as an integral part of the overall system (as opposed to evaluating it as a separate entity). It uses the same fundamental OT&E processes for MIS and embedded weapon systems software based on the premise that ***if the system works, the software works!***

NOTE: To be effective, software operational test planning must take place throughout the development process. Often, the developer and the SPO are reluctant to provide the operational tester with the documentation and materials needed to perform an effective evaluation of software maturity, reliability, supportability, usability, and effectiveness. Achieving cooperation among the system developer, the SPO, and the operational tester is an essential management prerequisite.

AFOTEC Lessons-Learned

AFOTEC has provided a list of lessons-learned based on the experiences of programs having completed the OT&E process.

- **Deputy for Software Evaluation (DSE).** A DSE should be assigned as early as possible to the SPO to become familiar with the system and to assist in detailed software OT&E planning. The DSE should be onboard at least 6 months prior to the first OT&E test event. (Larger programs may require even more lead time.) The

DSE, a software systems engineer, serves as the software evaluation team leader, is assigned to the test program, and coordinates and controls the completion of OT&E test plan objectives pertaining to software and software support resources.

- **Documentation.** The DSE and software evaluators must be provided current documentation and source code listings in time to perform evaluations. Promised deliveries not received can cause problems; therefore, it is to everyone's benefit to deliver requested documentation on time. You must also identify early the requirement for special sorties, equipment, and analysis support so that test requirements are accommodated.
- **Testing terminology.** Your team must come to terms with discussions over terminology and definitions (software faults, defects, errors, bugs, etc.). Industry-accepted definitions of software errors and defects (faults), found in the ANSI/IEEE, *Glossary of Software Engineering Terminology,* are listed in Table 14-4. [ANSI/IEEE83] A **failure** is an observable event (an effect). A **fault** is the cause of a failure. A fault may be caused by software or hardware. A software **defect** is the human-created element that caused the fault, such as those found in specifications, designs, or code. The bottom line with AFOTEC is, ***if an action is required of the operator due to a failure — it must be documented.***
- **Final test reports.** Striving for correct technical content, software test teams often write final test reports but are frustrated when OT&E headquarters personnel rewrite them for format and content. Time constraints and pride-of-authorship can strain tensions between the test team and headquarters. One solution is to have the two teams work together to review final report drafts early in the process. Also make sure that the report is written for a wide spectrum of readers, *computerese* is kept to a minimum, and that it is tailored for a senior officer audience with emphasis on results and recommendations.

NOTE: "*Timing*" is more important in real-time systems than in any other software development. The SEI has developed a method, Rate Monotonic Analysis (RMA), for managing the scheduling and execution of tasks for real-time systems. See Chapter 10, *Software Tools*, for a discussion on RMA and Volume 2, Appendix O, *Additional Volume 1 Addenda*, Chapter 10 Addendum B, "*Rate Monotonic Analysis: Did You Fake It?*"

CATEGORY	DEFINITION
Error	A discrepancy in implementing requirements or design specification. An error may manifest itself as incorrect or undesired results.
Fault	A defect in code that has the potential to cause (possibly invisible) incorrect or unexpected results. Faults are also known as bugs. Faults in code usually result in errors.
Debugging	The process of locating, analyzing, and correcting suspected faults.
Failure	The execution of a software fault or defect that manifests itself as incorrect or undesired results.
Testing	The process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.
Dynamic analysis	Testing by executing code.
Static analysis	The process of evaluating a computer program without executing it: e.g., review, desk check, inspection, or walkthrough.
Correctness	The composite extent to which: (1) Design and code are free from faults. (2) Software meets specified requirements. (3) Software meets user expectations.
Verification	(1) The process of determining whether or not the products of a given phase of the software development life cycle fulfill the requirements established during the previous phase. (2) Formal proof of program correctness. (3) The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements.
Validation	The process of evaluating software at the end of the software development process to ensure compliance with software requirements.

Table 14-4 ANSI/IEEE Software Engineering Terminology [ANSI/IEEE83]

IMPLEMENTATION

Implementation is where the software is **coded**, **unit tested** and **integrated** into a functional software product. The purpose of this phase is to put the design produced during the previous phase into practical effect. With **Ada** as the implementation language, this phase is relatively easy and is, in essence, an extension of the design process. The design phase produces a detailed design represented by a complete specification for a set of Ada program modules. During the coding phase, all that is required is that the modules (units) be

implemented. Of course, as units are coded, further decomposition may be required. In this regard, the design/code/test phases lose their distinction and should form an iterative process at each stage of the solution. Ada tools (such as syntax-directed editors) can increase programmer efficiency and software quality by making coding easier and by enforcing the creation of syntactically correct Ada modules. [BOOCH94]

BUILDING SECURE SOFTWARE

Security is an essential element of many major DoD software-intensive systems. Adversaries actively collect information about our new systems and software to negate their combat effectiveness and eliminate our advantage of surprise. You must actively plan for and apply OPSEC measures to protect crucial information throughout your acquisition process. The OPSEC process helps guide the development of OPSEC measures. The process asks: what needs to be protected, from whom, is there a potential for exposure of critical information, what are the risks, and how is protection to be accomplished? This team effort must be revisited as your program matures and parameters change. A well thought out plan of protection and its judicious application will ensure the integrity and combat effectiveness of new systems and help us attain our mission objectives.

Security Planning

Security is the crucial aspect of strategic system and software planning often overlooked. DoD contingency strategists (wargamers) envision the objectives of war in the 21st century, not as attacks to destroy enemy lives, but as maneuvers to gain control of those invisible, more vulnerable, more significant and consequential software-driven systems. Weapons systems dependent on satellite communications for target positioning, global financial systems, highly distributed military logistics and air traffic control systems, and secure telecommunication networks are *"soft"* because they are highly pregnable. [BLACK93] Due to its vulnerability, failure to plan for software security could prove catastrophic. Today, reports abound of hackers gaining unauthorized access to software systems, sometimes creating serious damage. Other software-related security problems have resulted in severe financial loss or even loss of life. ***The protection of your software must be a major element in your strategic planning process.***

The common objective of acquisition activities is the production of combat-ready weapon systems and/or support for those systems to further our national defense. The advantage we seek, the success of our defensive efforts, is often expressed in the element of surprise. *Surprise*, in this instance, means that our systems, when deployed, can operate in hostile environments and do the job for which they were designed. Lack of surprise means that an adversary already knows enough about our systems to counter them and/or to render them ineffective once deployed . Lieutenant General V.A. Reznichenko, authoritative tactician for former Soviet Union ground forces, explained why security is so important.

> *Surprise makes it possible to take the enemy unawares, to cause panic in his ranks, to paralyze his will to resist, to drastically reduce his fighting efficiency, to contain his actions, to disrupt his troops' control, and to deny him the opportunity to take effective countermeasures quickly. As a result, this makes it possible to successfully rout even superior enemy forces with the least possible losses to friendly forces.* [REZNICHENKO84]

Some program managers consciously omit security (and safety) requirements from their plans, as they believe such considerations will significantly increase software development costs. As in risk abatement, the benefits of including software security requirements upfront must be weighed against **life cycle costs**. Not planning for security upfront and having to address these requirements after development is underway (or the system is deployed), can severely impact the cost of your software development (and system life cycle costs) as they constitute significant **cost drivers**.

It is imperative that your new software (and hardware) be fully protected commensurate with your program requirements and sensitivities throughout the development life cycle to ensure it is fully combat effective at IOC and that the element of surprise is retained. It makes little sense to expend valuable resources (manpower, money, and time) on software that is compromised before it can fulfill design and mission objectives. Software protection must be an integral and normal part of all acquisition activities.

> *Power relationships among nations have changed. Increasingly, perceptions are focused on economic matters where onetime friends and allies are now viewed*

> *as competitors and even adversaries. Battlefields are now boardrooms. There is at least one constant currency of power, however, and that is information. Information processed into intelligence provides a basis for sound decision making in both military and corporate combat.* [PATTAKOS93]

Building preemptive defenses into your software is one way to fight the **software security** war against hackers and enemy access to our vital information resources. Another method is to build in the ability to bounce back quickly if penetration is accomplished. You must, therefore, plan for nonlethal warfare risks to be prepared, through prevention and circumvention, for today's *software-versus-software* battlefield.

NOTE: See Addendum A of this chapter, "*Multilevel Information Systems Security Initiative.*"

Operations Security (OPSEC)

OPSEC is specifically designed to control and protect information of intelligence value to an adversary. This information is called **critical information**. Critical information includes the specific facts about our intentions, capabilities, limitations, and activities needed by adversaries to guarantee the failure of our mission. It is the key information about our programs, activities, hardware, and software, which if lost to an adversary, would compromise that program. Critical information may be either classified or unclassified. It is not only the classification of the information that is important, but also the *value* of the information to an adversary. It requires the safeguarding of all classified information and protection from tampering for unclassified information.

OPSEC is implemented by the development of an **OPSEC Plan**. The plan is based on a thorough analysis of the important and sensitive aspects of your program (or software system) and of the environment for which the software is being developed. OPSEC planning follows the **OPSEC process**, a logical method of information analysis and evaluation guiding protection and control. The OPSEC process can be applied to virtually any software development activity, and is as simple or complicated as the situational environment warrants. The steps in the OPSEC process are:

- Identify critical information,
- Describe the intelligence collection threat,
- Identify OPSEC vulnerabilities,
- Perform risk analysis, and
- Identify countermeasures to control and protect the information.

The plan summarizes the results of this analysis process and becomes the framework for subsequent software protection measures.

Critical information. Because you have to know what to protect, the first step is to identify critical information and the **indicators** that point to or may divulge it. The first listing of critical information is in the **Operational Requirements Document (ORD)** developed by the user, which is very broad and general. As your program proceeds, this list must constantly be reviewed and refined. As your program matures, the list of critical information will become more specific and detailed.

Threat. The **threat** is specific information about an adversary's capabilities and intentions to collect critical information. It begins with the identification of the adversary(ies). The adversary's resources/ assets available to collect critical information and the degree of the intent to collect is then assessed. The threat assessment must be specific (e.g., geographical location, facility, program office, software system, laboratory, or contractor facility). Threat information must be obtained in coordination with the OPSEC officer or through local liaison officers and organizations (e.g., the Air Force Office of Special Investigations, Air Intelligence Agency, or the National Air Intelligence Center. Intelligence collection of threat information is also included in the **System Threat Assessment Report (STAR)** validated by the **Defense Intelligence Agency (DIA)**.

Vulnerability. Critical information and indicators of critical information are compared with the threat to determine if an OPSEC **vulnerability** exists. For an OPSEC vulnerability to exist, critical information must be potentially open and available to an adversary, and that adversary must have some type of collection platform in place to obtain the information (e.g., a spy satellite, an agent, an intelligence gathering ship, or communications network access). ***If sensitive information is available and an adversary can collect it, then an OPSEC vulnerability exists.***

Risk assessment. A **risk assessment** is a cost/benefits analysis of proposed protective measures and the mission imperative. Several factors drive this assessment. First, no system can be 100% secure unless it is sealed off from all outside influences. Second, whatever protective measures are used, they must not unduly hinder or prevent mission accomplishment or the attainment of program objectives. Finally, a balance must be found that provides the maximum possible protection while maintaining program integrity.

OPSEC measures. Various methods must be developed that best meet operational protection requirements while mitigating the identified OPSEC vulnerability. **OPSEC measures** are program specific and must be tailored to the identified vulnerability. OPSEC measures include:

- **Action control measures**. These are actions that can be executed to prevent detection and avoid exploitation of critical information. You should avoid stereotyped procedures which can be exploited by an adversary. Examples of action control include making preparations inside buildings rather than outside, conducting activities at night, and adjusting schedules or delaying public affairs releases.
- **Countermeasures**. These are methods to disrupt adversary information gathering sensors and associated data links or to prevent the adversary from obtaining, detecting, or recognizing critical information. Examples include jamming, masking, encryption, interference, camouflage, and diversions.
- **Counteranalysis**. These are methods to affect the observation and/or interpretation of adversary analysts. They do not prevent detection, but enhance the probability that the detectable activity is overlooked or its significance is misinterpreted. Counteranalysis measures provide uncertainty and alternative answers to adversary questions. Deceptions, including covers and diversions, are in this category of OPSEC measures. Detailed planning of deceptions are separate from protection planning. However, close coordination between OPSEC and deception planners will facilitate the desired result.
- **Protective measures**. These measures can and should include the use of all established security disciplines.

Although security is ultimately your responsibility, program protection is not a one-person job. OPSEC measures run the gamut of possibilities and there is ample help available. Each MAJCOM, product center, logistics center, test range, and laboratory has an identified OPSEC

point-of-contact. Indeed, each security discipline has a point-of-contact. **Software protection** is, thus, a coordinated team effort — the same as other program activities.

Historically, it has been difficult and expensive to design and build secure/trusted data systems. The traditional way of building secure systems has been to use logical and physical separation (i.e., an "*air gap*") based on providing a physically secure facility for each system, with everyone in the facility cleared to the level of the most sensitive data. This method is not only expensive, but very inefficient, and has several undesirable properties such as the cost of duplicating facilities, and multiple sets of hardware and software. There is also an inability to share personnel talent and skills due to the need for separation and number of expensive clearances for people who have no access to the data itself. Possibly the most serious issue is the inability to share data. This creates serious data concurrency problems as duplicated data in the myriad of systems are updated at different frequencies — greatly increasing the probability of error as the number of instances increases. This was a major problem during the **Gulf War**. Virtually all the data needed was in theater, but it was not accessible in a way that allowed coherent data fusion and integration.

This problem should soon be totally eradicated. All necessary COTS components for building operational systems [i.e., hardware/operating systems, networks, and **relational database management systems (RDBMSs)**] are **National Computer Security Classification (NCSC)** evaluated at the CB/B1 and B2 levels. The old quandary that "*COTS products are not secure*" and "*secure products are not COTS*" is no longer true. Today it is possible to get a hardware/ operating system-network-RDBMS combination that was evaluated together, which greatly reduces the accreditation effort of the developer and the user.

The RDBMS is the most critical portion of the secure solution. The first, and most important, concern should be a vendor's overall philosophy and commitment to developing secure products. Some build a minimally compliant product so they appear to have complete secure and non-secure product suites. Serious secure product vendors meet the extreme assurance requirements required at the B2 level and above, while others have layered C2/B1 level features that meet the minimal assurance requirements at B1 and below. Vendors who are serious about the secure products market also view security as an attribute of their product — not as a 150% to 200% premium over the price of their standard product.

Compatibility of the vendors' products at various levels is a major development security issue. Compatibility has many benefits such as the ability to partition data and applications across different levels without having to duplicate the applications. For example, the ability to access untrusted administrative systems and secure operational systems in the same application is useful. Also desirable is the ability to separate very sensitive SCI data into a B2 assurance RDBMS engine, routine operational data into a B1 assurance RDBMS, and other administrative data into a C2 assurance RDBMS. This makes data and security administration easier while retaining the usability and functionality of one logical database with joins and other transaction management capabilities. In addition to only having to develop one set of applications, this capability has several performance advantages. Joins are required only in those less normal scenarios where multiple kinds of data are required in a single transaction. Otherwise, a single server is used, increasing the apparent network bandwidth for users at different levels. The RDBMS' distributed capabilities should make the data partitioning invisible to the client-user so the only relevant issue is the client security level, not specialized knowledge of the physical data schema.

All secure products are evaluated against the **Trusted Computer System Evaluation Criteria (TCSEC)** defined in DoDD 5200.28 (the Orange Book) and its various interpretation guides. The **Orange Book** is a statement of the DoD basic security policy and relies on the **Bell-Lapadula Security Policy Model**. The principle of the Bell-Lapadula model is access mediation based on the relative values of a user's (subject) clearance level and the data (object) classification level as conveyed by appropriately assigned security labels. The salient features of the model are a "*subject*" may access "*objects*" at its session (login) level and below, and "*may-write objects*" at its session level only.

This policy has some onerous implications for RDBMS.' The most serious of which are that implementation of this policy dictates that: (1) UNIQUEness of a primary key is only guaranteed within a single security level; (2) an index on a table exists at a single security level; and (3) referential integrity is guaranteed only at a single security level. Related issues are the serious covert channels in databases centered around the physical storage of labels in each row and the serialization of row IDs. The management of data integrity locking mechanisms at different security levels is also a problem. (INFORMIX uses a unique security metadata approach which eliminates all these covert channel issues by avoiding the need to physically store labels

in each row.) To support complex and sophisticated application development, most secure RDBMS' provide a means for mitigating these problems.

The crucial item is the safety and granularity of these mechanisms. Most secure RDBMS' support the simple, but coarse, method of using a configuration option to set it to either "*on*" or "*off*." A more sophisticated approach is to support a set of discrete privileges granted and revoked selectively by the **System Security Officer (SSO)** to facilitate a specific task. These discrete privileges are manageable at a granularity no greater than a transaction boundary, and deal with the granularity of indices, uniqueness of primary keys, referential integrity across levels, and locks at multiple levels.

The selection of a secure RDBMS should not lock a developer into a particular hardware environment. A committed secure products vendor will support mainstream hardware platforms and operating systems (e.g., HP, Sun, IBM, DEC, Harris, AT&T, and SCO). They will also support all applicable standards [such as FIPS 127, FIPS-156, XOPEN, RDA, ANSI XXX, and *de facto* standards (e.g., DRDA, ODBC, and TCP/IP)] in their standard and secure products. A secure product should not have a significant performance degradation over an equivalent non-secure product. Vendors should publish official audited benchmarks of both secure and non-secure products. An example of such a product is the INFORMIX Online Secure Product. *[See Volume 2, Appendix A for information on how to contact INFORMIX.]*

NOTE: See US Department of Commerce, NIST Special Pub 800-7, *Security in Open Systems*, July 1994.

SOFTWARE DOCUMENTATION

Documentation must support the entire life cycle. Most importantly, it provides fundamental guidance for PDSS. Documentation can be categorized as being either technical or managerial. **Technical (or engineering) documentation** is necessary as it records the engineering process and helps software engineers know where they are, what they have produced, and how. It also helps maintainers and other engineers understand the code developed by others. **Management documentation** is that produced for the Government or the development organization to aid in monitoring the contractor's

development progress in achieving program milestones and in fulfilling performance requirement specifications.

Although it often represents the foundation of a successful software development, documentation can also represent a significant source of cost and schedule risk. Perhaps the most decorated, colorful, and outspoken US Marine in history, Lt. General Lewis B. "Chesty" Puller, condemned documentation when he stated: "*Paperwork will ruin any military force.*" [PULLER62] Not only will it ruin any military force, ***paperwork will ruin any software development.*** Overloading your contractor with excessive documentation requirements takes away from engineering activities — costing valuable time and money.

Conversely, too few requirements for technical documentation may cause loss of program visibility and control. **Design documentation** that does not reflect the delivered software's current state is worse than no documentation at all. It translates into high maintenance costs when attempts to enhance or upgrade the system are hampered by insufficient information on the delivered source code. Allocating operational functional requirements to configuration items should be both a management and a technical decision as it establishes the framework for collecting information on software requirements, design, code, and testing. ***Shortcuts on maintaining/updating technical documentation should be avoided at all costs.*** Whenever the developer makes changes to data flow, the design architecture, module procedures, or any other software artifact, supporting **technical documentation** must be periodically updated to reflect those changes. This requirement must be clearly stated in the contract CDRLs. No matter how well your software product performs in fulfilling user requirements, if its supporting technical documentation is inadequate, your system is not a quality product. Without quality documentation the product can neither be adequately used nor maintained. Documentation (either in paper copy or electronic format) that confuses, overwhelms, bores, or generally irritates the users is of little or no value to you or your customers. [DENTON92]

NOTE: Consider requiring on-line access to the developer's software engineering environment as an alternative to technical documentation.

Documentation is one of those activities that requires experience to determine a proper balance between too much and too little. Too little **technical documentation** can create the proverbial *"maintenance man's nightmare;"* whereas, too much effort expended on producing unnecessary **management documentation** can waste precious development time and dollars. Table 14-5 illustrates the difference in time spent by software developers fulfilling DoD contracts and in fulfilling commercial software contracts. Management, meeting support, and documentation requirements comprise 45% of the total effort for military software developments, as compared to the 22% of effort spent on the same activities for commercial applications. Compared to the commercial sector, DoD places more emphasis on the management burden than on engineering. As you learned in Chapter 4, *Engineering Software-Intensive Systems*, a high concentration on upfront software engineering is essential to ensure that quality, supportability, and usability are built-in.

	MILITARY SOFTWARE	COMMERCIAL SOFTWARE
Engineering	30%	50%
Evaluation	20%	20%
Management	15%	10%
Meeting Support	15%	5%
Documentation	15%	7%
Customer/User Support	5%	8%

Copyright by RCI

Table 14-5 Military/Commercial Effort Distribution

Must-Have Documentation

Even where program management documentation is kept to a minimum, **management** and **quality metrics reporting** is essential and should be a contract requirement. Metrics reports describe the contractors' progress against their plan. They reveal trends in current progress, give insights into early problem detection, indicate realism in plan adjustments, and can be used to forecast future progress. If not required, software developers are often reluctant to commit to paper their deficiencies and/or accomplishments. Your contractor may be agreeable to your suggestions and direction early in the game.

However, as the development progresses and problems are encountered, this agreeability can deteriorate and the contractor may increasingly ask, *"Where in the contract (or other documentation) does it say the software has to do that?"* Once the honeymoon is over, the *documented word* (either in the contract, through delivered metrics documentation, or on-line access) has the most influence on contractor actions. By stressing the importance of metrics reporting early, you can avoid many problems later on.

From the user's perspective, the software is only as good as the documentation (both written and interactive) that tells them how to use it. Failure to include in the **user's documentation** changes made before delivery to the executable software can have profoundly negative effects. For example, changes in the order or format of the interactive input to an MIS, if not documented, can cause significant problems through confusing error messages — or even system crashes.

Software documentation can provide a vehicle for ***early user involvement*** in the development process. User visibility is necessary to ensure that user requirements are addressed early, rather than added later at much greater expense. Specification and design documents give the user the opportunity to review requirements before the system is designed or coded. If user documentation is not available for user review until program completion, design changes resulting from that review can cause significant schedule slippages and additional costs. Software requirements and designs must be clearly documented so they can be evaluated and deficiencies can be discovered and corrected. Remember, the best software design is of little value if it is incomprehensible to those who must translate it into code.

NOTE: *Touch-and-feel* demonstrations are more effective mediums for user review of requirements than written specifications and design documents.

Technical documentation should never be produced after the fact, nor for bureaucratic reasons. Like metrics, documentation must be an outgrowth of the normal development process, not an end in itself. It must be produced to capture the engineering process so that you and others can understand and benefit from what has occurred. Clear documentation prevents developers from getting lost in production activities, and helps maintainers in understanding what the software does and how it was built. Documentation must be prepared throughout

the development process to capture the results of each engineering activity. [MARCINIAK90] Documentation, used as a reference tool to aid new personnel, users, and maintainers in becoming familiar with the software product, must be kept up to date. If not kept current, it will impede operational and maintenance efforts resulting in a needless waste of time, effort, and money. (Robust **Ada code** with its narrative characteristics is almost self-documenting. However, the architecture and concept of operations must be clearly described.) *[See the STSC's Documentation Technology Report, April 1994.] [On-line access to (or delivery of) technical documentation in electronic format saves development time and dollars.]*

CAUTION! The time and money saved by delaying or foregoing immediately needed documentation actually wastes time and money later in the program.

The bottom line for successful software development is adherence to the ***software engineering discipline*** discussed throughout these Guidelines for its stabilizing effects on the development process. No sounder advice can be given. As General George Washington explained in a letter of instructions to the captains of his Virginia regiments in 1759,

> *Discipline is the soul of an army. It makes small numbers formidable; procures success to the weak and esteem to all.* [WASHINGTON59]

	COMMANDMENT	WHERE DISCUSSED IN GUIDELINES
1	Thou shalt use risk assessment to direct the development strategy.	Chapter 6, *Risk Management*
2	Thou shalt create and maintain a Software Development Plan with quantifiable progress indicators.	Chapter 8, *Measurement and Metrics*
3	Thou shalt manage the specification and change of requirements.	Chapter 15, *Managing Process Improvement, "Configuration Management"*
4	Thou shalt develop a Test Strategy and Plan as part of the initial planning process	Chapter 14, *Managing Software Development*
5	Thou shalt use metrics to continually assess both process and product.	Chapter 8, *Measurement and Metrics*
6	Thou shalt have a designated software technical lead.	Chapter 1, *Software Acquisition Overview, "People"*
7	Thou shalt conduct formal inspections.	Chapter 15, *Managing Process Improvement*
8	Thou shalt identify and track discrepancies throughout the entire life cycle.	Chapter 8, *Measurement and Metrics*
9	Thou shalt not select CASE tools before establishing methods	Chapter 10, *Software Tools*
10	Thou shalt not let documentation drive the software development process	Chapter 14, *Managing Software Development*

Table 14-6 The Ten Commandments of Software Development [DYE93]

REFERENCES

[AGRESTI86] Agresti, William W., ed., New Paradigms for Software Development, IEEE Computer Society Press, Washington, D.C., 1986

[BESIER95] Besier, Boris, Black-Box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, Inc., New York, 1995

[BLACK93] Black, Peter, "The Next Generation of Weapons: Dependency on Electronic Systems Make Us Vulnerable," *Washington Technology*, December 2, 1993

[BOEHM76] Boehm, Barry W., "Software Engineering," *IEEE Transactions on Computers*, C-25, No. 12, December 1976

[BOOCH94] Booch, Grady, Software Engineering With Ada, Third Edition, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1994

[BROOKS75] Brooks, Frederick, The Mythical Man-Month: Essays on Software Engineering, Addison Wesley, Reading Massachusetts, 1975

[BRYKCZYNSKI93] Brykczynski, Bill and David A. Wheeler, "An Annotated Bibliography on Software Inspections," Institute of Defense Analysis, Alexandria, Virginia, January 1993

[CAID91] Government/Industry Acquisition Process Review Team, *Clear Accountability in Design*, Final Report, October 1991

[CHANNING92] Channing, William Ellery, as quoted by Lowell Jay Arthur, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993

[COAD90] Coad, Peter and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[CUSUMANO95] Cusumano, Michael A., and Richard W. Selby, Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People, The Free Press, New York, 1995

[DEMING82] Deming, W. Edward, *Out of Crisis*, Massachusetts Institute for Technology, Center for Advanced Engineering Study, Cambridge, Massachusetts, 1982

[DENTON92] Denton, Lynn and Jody Kelly, Designing, Writing & Producing Computer Documentation, McGraw-Hill, Inc., New York, 1992

[deSAXE32] de Saxe, Field Marshall Maurice Comte, My Reveries on the Art of War, 1732

[DISNEY76] Disney, Walt, as quoted by Bob Thomas, Walt Disney, An American Original, Simon & Schuster, New York, 1976

[DMR91] "Strategies for Open Systems," briefing presented by DMR Group, Inc. to SAF/AQK, March 14, 1991

[DOBBINS92] Dobbins, James H., "TQM Methods in Software," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[DYE93] Dye, Dick, "The Ten Commandments of Software Development," *CTA Tech-News*, June 1, 1993

[HEIKAMP94] Heitkamp, Kenneth B., "Software Development Process Changes Implemented by the Cargo Movement Operations Systems (CMOS)", memorandum from SSC/EA to SAF/AQK, February 25, 1994

[HOROWITZ91] Horowitz, Barry M., "Architecture, Architecture, Where Art Thou, Architecture?" briefing prepared by MITRE, May 8, 1991

[JONES86] Jones, Capers, Programming Productivity, McGraw-Hill Book Co., New York, 1986

[KEENE91] Keene, Charles A., white paper "Lessons-Learned: Nuclear Mission Planning and Production System," AF Strategic Communications-Computer Center (SAC), Offutt AFB, Nebraska, January 17, 1991

[KELLER93] Keller, Ted, briefing "Providing Man-Rated Software for the Space Shuttle," IBM, Houston, Texas, 1993

[KOLKHORST88] Kolkhorst, Barbara G., and A.J. Macina, "Developing Error-Free Software," Fifth International Conference on Testing Computer Software, US Professional Development Institute, Silver Springs, Maryland, June 1988

[McMENAMIN84] McMenamin, Steve and John Palmer, Essential Systems Analysis, Yourdon Press, Englewood Cliffs, New Jersey, 1984

[MERRILL92] Merrill, Paul H., Not the Orange Book, Merlyn Press, Wright-Patterson, AFB, Ohio, 1992

[MOSLEY93] Mosley, Daniel J., The Handbook of MIS Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing, Yourdon Press, Englewood Cliffs, New Jersey, 1993

[NAPOLEON08] Napoleon I, a conversation with Marshall Murat on March 14, 1808, Christopher J. Herold, ed., The Mind of Napoleon: A Selection from his Written and Spoken Words, Columbia University Press, New York, 1955

[PAULSON79] Paulson, Paul J., as quoted in the *New York Times*, May 4, 1979

[PAYTON92] Payton, Teri F., briefing, "Reuse Context," presented at the STARS/Air Force Reuse Orientation, October 14, 1992

[PRESSMAN93] Pressman, Roger S., "Understanding Software Engineering Practices: Required at SEI Level 2 Process Maturity," briefing prepared for the Software Engineering Process Group, July 30, 1993

[PUTNAM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software On Time, Within Budget, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992

[REZNICHENKO84] Reznichenko, Col General V.G., Taktika, 1884

[SHUMATE92] Shumate, Ken and Marilyn Keller, Software Specification and Design: A Disciplined Approach for Real-Time Systems, John Wiley & Sons, Inc., New York, 1992

[STRASSMANN93] Strassmann, Paul A., "Information Warfare for Low-Intensity Conflicts," briefing presented to the Army Executives for Software (ARES), West Point, New York, July 15, 1993

[THOMPSON91] Thompson (SCXS), "Guidelines" comments from SCXS, March 15, 1991

[WASHINGTON59] Letter to the captains of the Virginia regiments, July 29, 1759, The Writings of George Washington, John C. Fitzgerald, ed., Washington DC, 1931-41

[YOURDON90] Yourdon, Edward N., Modern Structured Analysis, Prentice Hall, New Jersey, 1990

[YOURDON92] Yourdon, Edward N., Decline and Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992

CHAPTER 14
Addendum A

The Multilevel Information Systems Security Initiative

Robert Cooney
Gloria Bilinski

EDITOR'S NOTE: *This article appeared in the Navy newsletter, CHIPS, July 1995*

THE NEED FOR INFORMATION SECURITY: Providing the Basic Building Blocks for Computer and Communications Security

Computer systems and networks are playing an increasingly significant role in the everyday activities of DoD workers. Daily communications take place through an increasing number of applications, such as e-mail. Files are transferred and databases are accessed to obtain information. Users even login to computer systems from remote locations while on travel or other duty assignments.

How does this happen? Connectivity to either a LAN, such as Banyan or Novell, or a wide area network (WAN), such as the DISN, NAVNET or Internet, provides the travel path (the Information Highway), for information and communications to be exchanged locally, nationally and worldwide.

However, the increased usage of computer systems and networks in daily communications and other work-related activities brings with it certain concerns. One of the most important concerns is a need for security. A typical example is the use of e-mail. When it's used for

important business transactions and organizational communications, rather than just interpersonal messages, additional security requirements begin to arise.

Certain Assurances Are Needed

- Information exchange with other users is authentic — it has not been modified or tampered with.
- Information received originated from valid, authorized parties.
- Information sent is received only by the intended parties.
- Privacy — information exchanged cannot be observed by unauthorized viewers.
- Any information or data has definitely reached its destination.
- Assurances and protections should not only be reliable and dependable, but also cost-effective.
- Because of the increased use of commercial-off-the-shelf products, users also want security protections capable of being integrated into commercially available products.

While this may seem like a tall order, it truly represents the needs of DoD's emerging network cruising employees.

THE MISSI SOLUTION

In response to these needs and requirements, the National Security Agency (NSA) began a computer security development effort called the Multilevel Information Systems Security Initiative (MISSI). MISSI encompasses both the traditional Communications Security (COMSEC) and Computer Security (COMPUSEC) disciplines. MISSI's goal is to provide dependable and affordable security services necessary to protect information from unauthorized disclosure or modification and to provide mechanisms to authenticate users participating in the exchange of information.

MISSI will provide, in the not-too-distant future, multilevel security (MLS) technology — the ability to combine automated information systems (AISs) of different classifications or sensitivity levels into one single, integrated system. In today's networking environment, more and more computer systems are being consolidated. As different networks are integrated, and different levels of information are being handled on a network, each level must be protected sufficiently. MLS permits different levels of information, such as unclassified, sensitive and secret to be managed and controlled, so that although there is a

single, integrated system, the information will still be separated according to its classification. Users will be able to integrate different classifications of information, with adequate security protection being provided to the full range of security levels.

The MISSI Approach

To provide security services, MISSI is evolving a series of products designed to be as flexible as possible. NSA worked closely with users to determine their security needs. After the requirements were identified, they were incorporated into the MISSI products. NSA also worked with industry so the products developed would be based on common standards and interoperable with commercially available products. MISSI's products will evolve incrementally, as new requirements are identified and new technology becomes available. Each product release will increase the security protections available to users, but will still be compatible with preceding releases.

MISSI will provide basic building block products. These products, when combined, will provide security for computer applications such as e-mail systems, file transfer activities, database operations and remote logins. MISSI will also include a security management infrastructure so that security services can be managed.

MISSI Release 1

The first release of MISSI will focus on protecting sensitive but unclassified (SBU) e-mail in the Defense Message System (DMS). DMS is an important part of the Defense Information Infrastructure (DII) and will be implemented in strategic, tactical, fixed and mobile environments. In fact, all electronic messaging within DoD will have to migrate to DMS-compliant messaging. MISSI's first release centers on providing security at a user's workstation with writer-to-reader or desktop-to-desktop protection for DMS users. Collectively, MISSI products will provide the following security services:

- **Data integrity**. Data integrity is the assurance that no changes have been made to information that has been sent. Users can be confident that data sent is the same data that is received, that no unauthorized modifications have been made.
- **User identification and authentication**. Authentication services verify the identity of the creator or originator of a message. The recipient of the message can be certain that the sender is the named originator and not some impostor or other fraudulent entity.

- **User non-repudiation.** Non-repudiation services provide undeniable proof of the identities of both the originator and the recipient of a message. Neither party can deny involvement in the information exchange.
- **Data encryption and decryption.** Protection is provided through data encryption and decryption mechanisms allowing messages to be confidential and private. Only the intended recipient has access.

The Fortezza Card

The star of the show is the product used to provide these services, a cryptographic card known as the Fortezza card. Originally, this crypto card was called the Tessera crypto card, but because of a copyright infringement, that name was changed. Fortezza is an Italian word which, when translated into English, means a fortress. Based on the Personal Computer Memory Card International Association (PCMCIA) industry standard package, the Fortezza card is a combination of hardware and software. Although it is only the size of several credit cards stacked together, it is, in fact, a separate computer on a card. It contains its own processor and memory, and inputs and outputs through the 50 pin connection points on the end of the card.

DISA plans to give every DoD employee who will be issuing DMS messages into the DISN a Fortezza crypto card. Along with cryptographic data and various MISSI algorithms, the card will contain important security information about the user to which it belongs, such as the user's credentials, clearance information and authorizations.

To use the Fortezza card, it must be inserted into a special card reader installed on the user's workstation or PC. The Fortezza card will work with DOS, Windows, SCO UNIX, Sun OS, Solaris, HPUX and Macintosh operating systems. The card is activated by a four-digit personal identification number (PIN), similar to those used at automatic teller machines.

The Fortezza card uses public key technology which features a unique public key and private key. Cryptographic functions are performed using a private cryptographic key stored on the card that is unique to the card's owner. The private key will be kept secret by the card and should be used only by the owner of the Fortezza card. The public key should be made available to everyone. Although the two keys are

mathematically related, the private key cannot be determined from the public key. For DMS, the public keys will be available in the DMS X.500 Directory. This should quickly expand into the DoD Directory and eventually be used by applications other than messaging. (See earlier articles in *CHIPS.*)

The cryptographic functions allow a user to encrypt and decrypt e-mail, giving the user privacy and confidentiality. The Fortezza card also allows a user to digitally sign messages, replacing a handwritten signature. The digital signature security service is the electronic equivalent of registered mail. Just like a handwritten signature, a digital signature can be used to identify and authenticate the originator of a message verifying a message was sent by a particular user. A digital signature can also confirm that information has not been changed or modified after it was signed, thus ensuring message integrity.

Although the first release of MISSI will handle sensitive but unclassified messaging, future releases will support stronger cryptography and will be used to protect classified information. Watch for additional articles on Fortezza Plus and Fortezza for Secret, two follow-on programs of NSA that use this technology.

MISSI Security Management Services

The first phase of MISSI includes the infrastructure necessary to manage the security services provided by MISSI. This infrastructure will generate, distribute, update and revoke the cryptographic keys and the Fortezza cards. Specifically, security management involves managing the cryptographic keying, access control permissions and digital signature mechanism. It will also collect and analyze audit data relevant to security.

These security management functions will be performed on a workstation with special purpose application software. After the most recent change in component names, this workstation is now called the Certification Authority Workstation (CAW). This unique component, which will actually program Fortezza crypto cards, will be deployed in the more densely populated locations and, eventually, will most likely be found at the command level.

Other MISSI Products and Applications

Besides the Fortezza card, the MISSI building block products include firewall products, Secure Network Server (SNS) products and in-line network encryption products. A **firewall** is a set of components used to control access between two networks. Firewalls protect a network because they are designed so all traffic must pass through the firewall components. Only authorized computer communications and traffic is permitted to pass. Use of proxy agents and packet screening by both the sender and recipient address blocks provides most of the real work of this device.

A **Secure Network Server** is a computer that typically resides on the local network boundary acting as an MLS guard for the information handled and transmitted on the local network. An SNS located on the border of a computer network handling Secret information is an example. The SNS would ensure that Secret information handled on the network being protected would not inadvertently be passed to a network that is not equipped to handle Secret data, such as an Unclassified network.

In-line Network Encryption (INE) products are usually located at enclave boundaries between local and wide area networks, or on a single network between individual hosts/workstations that are operating at different security levels. These products will provide both encryption and access control services. By providing end-to-end encryption of data communications and access control between local area networks, INE products will ensure that information being transmitted is not disclosed to unauthorized parties.

Applications other than e-mail may benefit from MISSI products. Authenticated logons, CD-ROM encryption, fax encryption, electronic commerce and electronic data interchange can use the encryption and digital signature capabilities provided by MISSI.

If you are interested in obtaining additional information about any of the components mentioned, give us a call. We have a MISSI Testbed to demonstrate these components and can quickly prototype your own unique environment. Several pilots are currently in place at the Washington Navy Yard.

CHAPTER 14
Addendum B

If Architects Had to Work Like Programmers

Mike Morgan
PKR2, Defense Information Systems Agency

Dear Mr. Architect:

Please design and build me a house. I am not quite sure what I need, so let's get started. My house should have between two and 45 bedrooms. Just make sure the plans are such that the bedrooms can be easily added or deleted. When you bring the blueprints to me, I'll make the final decision about what I want. Also, bring me the cost breakdowns for each configuration so I can arbitrarily pick one at a later time.

Keep in mind that the house I ultimately choose must cost less than the one I am currently living in. Make sure, however, that you correct all the deficiencies that exist in my current house (the floor of my kitchen vibrates when I walk across it, and the walls don't have nearly enough insulation in them).

As you design, also keep in mind that I want to keep yearly maintenance costs as low as possible. This should mean the incorporation of extra-cost features like insulated windows or composite siding. (If you choose not to use Anderson insulated windows, be prepared to explain you decision.)

Please take care that modern design practices and the latest materials are used in construction of the house, as I want it to be a showplace for the most up-to-date ideas and methods. Be alerted, however, that

the kitchen should accommodate (among other things) my 1952 Gibson refrigerator.

To assure that you are building the correct house for our entire family, you will need to contact each of my children and our in-laws. My mother-in-law will have very strong feelings about how the house should be designed, since she visits us at least once a year. Make sure you weigh all these options carefully and make recommendations. However, I retain the right to overrule any recommendation you make.

Please don't bother me with small details right now. Your job is to develop the overall plans for the house and get the big picture. At this time, for example, it is not appropriate to be choosing the color of the carpeting; however, keep in mind that my wife likes blue.

Also, do not worry at this time about acquiring the resources to build the house itself. Your first priority is to develop detailed plans and specifications. Once I approve these plans, however, I would expect the house to be under roof within 48 hours.

While you are designing this house specifically for me, keep in mind that sooner or later I will have to sell it to someone else. It should — therefore appeal to a wide variety of potential buyers. Please make sure, before you finalize the plans, that there is a consensus of the potential home buyers in my area that they like the features of this house.

I advise you to run up and look at the house my neighbor built last year, as we like it a great deal. It has many things that we feel we need in our new home, particularly the 75-foot swimming pool. With careful engineering, I believe you can design this into our new house without impacting the construction cost.

Please prepare a complete set of blueprints. It is not necessary at this time to do the real design, since they will be used only for construction bids. Be advised, however, that you will be held accountable for any increase of construction cost as a result of later design changes.

You must be thrilled to be working on such an interesting project! To be able to use the latest techniques and materials and to be given such freedom in your designs is something that can't happen very often. Contact me as soon as possible with your ideas and completed plans.

Sincerely,
The Client

PS: My wife just told me she disagrees with many of the instructions I have given you in this letter. As the architect, it is your responsibility to resolve these differences. I have tried in the past and have failed to accomplish this. If you can't handle this responsibility, I will have to find another architect.

PPS: Perhaps what I need is not a house at all, but a travel trailer. Please advise me as soon as possible if this is the case.

CHAPTER 14
Addendum C

On Board Software for the Boeing 777

NOTE: **This article is found in Volume 2, Appendix O, Additional Volume 1 Addenda.**

CHAPTER

15

Managing Process Improvement

CHAPTER OVERVIEW

General Edward C. Meyer, former Army Chief of Staff (1979-1983), had good advice for software program managers.

> *Leadership and management are neither synonymous nor interchangeable. Clearly good civilian managers must lead and good military leaders must manage. Both qualities are essential to success.* [MEYER80]

In this chapter you will ***learn that managing process improvement*** *requires a commitment from you to provide leadership, management direction, and the resources necessary to achieve improvement goals. The theory and concepts behind process improvement, such as* ***building-in-quality*** *through prevention of errors and early detection of inserted defects, perpetual improvement, and customer focus, are all aimed at the software development effort. Technology, such as the TQM process of Plan, Do, Study, and Act are the tools by which quality objectives can be achieved. It is your job to institute process improvement throughout your program by ensuring the software process is constantly revisited, analyzed, and improved throughout its life cycle. Your development team must be motivated, mobilized, trained and supplied with the resources necessary to accomplish their process improvement goals.* ***Clearly, your commitment is the most vital element. Without it, process improvement is not attainable.***

Process improvement is a never-ending task that can be accomplished through an effective and interactive development process, quality built-in and controlled, metrics, reviews, audits, and inspections, proactive, meaningful training, and well-planned and executed configuration management. Another key factor in managing process improvement is feedback through disciplined program/contract measurement and monitoring. One approach for tracking contractor performance is to implement an earned-value/credit/revenue method for accepting and rating deliverables. ***Earned-value*** *is tied to objectively observable steps, deliverables, and milestones. Basing your management approach on these activities will give you the means to make quality assessments, establish a proactive decision process, and attain your quality goals.*

Defects are one of the greatest risk items in software development. A commitment to software quality demands that defects be eliminated, not only

from all products delivered, but from the processes that create those products. Sources of errors must be identified with emphasis placed on improving techniques for their early removal. Sound software engineering includes multiple embedded mechanisms for early defect causal analysis with ***defect prevention*** *as the goal. Every new build should contain less defects than its predecessor as your software evolves into a zero-defect product.*

Software quality assurance *teams, process action teams, reviews, audits, and peer inspections are all geared towards the zero-defect challenge. They should be integrated into a well-rounded total quality management process. Peer inspections are an especially effective quality assurance tactic as they lead to improved product quality by instilling a sense of teamwork and pride in producing quality work. Independent verification and validation has traditionally been a necessary component for all major software developments. An IV&V team, established as part of the program team and composed of contractors or independent DoD evaluators, may still be useful; but, it may be unnecessary if truly effective process control with strong peer inspection mechanisms is in place.*

Cleanroom engineering *is a statistically controlled development method based on a team-oriented process of certification for reliable software systems. This method solves the unit testing problems of programmer bias and testing coverage. In Cleanroom, correctness is built into the software by the development team through special techniques for specification, design, and verification, prior to its release to Cleanroom certification test teams. This replaces traditional unit testing and debugging, and produces software of sufficient quality to directly enter system testing with no prior execution. All errors are accounted for from first execution on—* ***with no private debugging allowed****. Cleanroom software has typically entered system testing with near to, or zero, defects. Peer reviews and a concentrated team effort to produce zero defects contribute to the dramatic improvements in quality experienced by Cleanroom teams.*

Increasing productivity is a paramount process improvement goal. Costs go down, quality goes up, and schedules shrink when productivity is improved. Ada, reuse, design and process simplicity, end-user involvement, prototyping and demonstrations, and sophisticated automated tools are all productivity enhancers. Another way to increase productivity is by selecting the contractor with the most mature development capability, exceptional skills and experience, and a history of success in developing Ada software in your domain of comparable size, scope, and complexity. Configuration management is another vital factor in successful software development. It is the glue that holds the whole development together. It keeps track of where you have been, where you are now, and where you are going. The earlier the contractor establishes configuration management control over executable code, the greater the probability for success.

CHAPTER 15

Managing Process Improvement

ATTAINING THE QUALITY OBJECTIVE

> *I hold that leadership is not a science, but an art. It conceives an ideal, states it as an objective, and then seeks actively and earnestly to attain it, everlastingly persevering, because the records of war are full of successes coming to those leaders who stuck it out just a little longer than their opponents.*
> —General Mathew B. Ridgway [RIDGWAY66]

The concept of **process improvement** is not a panacea, a quick fix, a passing fancy, nor a trendy management buzzword. It is a framework from which an ideal state can be approached. It places your program in a state of constant improvement to produce customer-defined quality products. Former Under Secretary of Defense for Acquisition, **Robert B. Costello**, defined process improvement as

> *...not a finite program with a beginning and an ending. It must be woven into the fabric of a management style. It must be built into the way we do our day to day business...[It] is not a vague concept, nor a program. It's a managed, disciplined process for improving quality, increasing productivity, and eliminating non-value added activity. From a conceptual viewpoint,...quality management makes the top manager squarely responsible for the quality of the organization.* [COSTELLO88]

By requiring that offerors describe their approach to **process improvement** in their proposals, you will gain insight into whether

they can and will produce quality software. If there is a corporate commitment to process improvement, there is a better chance they will have control over the critical processes used to develop their software. Control of critical processes leads to a predictable, repeatable, measurable development process. This, in turn, improves the quality of the software products—the outputs of those processes. [BAKER92] By requiring in your RFP a software **Process Improvement Plan**, you will be including an important risk management element that treats the software task as a process that must be controlled, measured, and improved. Process improvement activities are also critical in determining an organization's software development maturity level. [BAKER92] The **DoD Total Quality Master Plan** clearly states, "*source selection strategies will consider continuous process improvement as one element of selection.*" [TQMMP88] Process improvement not only increases software quality, but also increases productivity — resulting in shorter development times and cost savings.

WARNING! Many contractors can write good plans, but have a poor track record at executing them. In addition to a good plan, insist that contractors provide demonstrable evidence that they have a process improvement plan in place and that real improvement has resulted over time. Beware of *"something new just for you!"*

A software process problem usually translates into a software quality problem. The problem is often an ill-defined or missing process resulting from unrealistic schedules or insufficient resources. Whatever the problem, quality is always increased through effective process improvement. **Managing for process improvement** is a technique for achieving increasingly higher levels of quality, schedule reductions, higher productivity, and lower product costs. Because improvements do not stand everlasting once a quality goal is achieved, you must throw out the old adage, "*If it ain't broke, don't fix it.*" Your leadership must stress that even if something appears to be working, it can always be made better. Unremitting improvement demands that you fervently, passionately, and forever challenge and upgrade your program status quo.

DoD has made a commitment to **Total Quality Management (TQM)**. Accomplishing TQM objectives within your program depends on the application of concepts, theory, technology, people,

planning, and organization. Process improvement cannot be accomplished without a **Process Improvement Plan**. This plan, which should be a required deliverable in the RFP, must specify a vision and short-term, mid-range, and long-term improvement goals.

Continuous improvement focuses on management control, coordination, and feedback from three concurrent processes: (1) the software development process, (2) the error/defect analysis and prevention process, and (3) the quality improvement process. [KINDL92] Successful software managers establish procedures to incessantly improve these processes through the use of quality assessments and management techniques to attain quality goals.

NOTE: See the discussion on error/defect detection and removal in Chapter 14, *Managing Software Development.*

The long term benefits of process improvement are significant. Quality procedures reduce the number of errors/defects inserted during each phase of software development. This translates into a decrease in scrap and rework and greater efficiency for all subprocesses. Rather than correcting errors and defects, developers concentrate on preventing them through careful work, improved communications, control of procedures, and satisfaction of customer needs. ***A sound metrics program is the foundation of a quality process.*** Metrics are not, however, just a technique for assessing quality. They must be applied to program management and engineering activities as well. They play the most important role in evaluating the processes and products of your development efforts. [MARCINIAK90] *[See the STSC's report, Process Technology, Volume 1, March 1994.]*

The practice of process improvement has other positive impacts. It stabilizes development with repeatable processes. It defines clear procedures for change and enables gradual technology transition. There is less resistance to new technology because the implementors of change are the same people who suggest it. At the very least, process improvement fosters a willingness to try new ideas. [KINDL92]

Quality management concentrates on all repetitive, cyclical, or routine work, and its improvement. It defines processes, process owners, requirements for those processes, measurements of the process and its outputs, and feedback channels. [ARTHUR93] Deming's *"chain-reaction-theory"* states that ***improvements in***

quality always and automatically result in reductions in cost and schedule with increases in productivity and performance. He stressed that choosing quality does not result in tradeoffs in the other success discriminators. To manage process improvement, Deming proclaimed, you must "*adopt and institute leadership.*" [DEMING82] Management concerns in which you can institute leadership and directly influence relentless process improvement include:

- Risk management *[discussed in Chapter 6, Risk Management],*
- Program/contract management,
- Error/defect detection, removal *[discussed in Chapter 14, Managing Software Development],* and prevention,
- Process/product measurement *[discussed in Chapter 8, Measurement and Metrics],*
- Software reviews, audits, and peer inspections,
- Independent verification and validation,
- Reuse *[discussed in Chapter 9, Reuse]*
- Productivity *[discussed in Chapter 8, Measurement and Metrics],* and
- Configuration management.

A **process-focused approach** can achieve progressive, measurable improvement in your program. *"Process-focused"* means that your attention is concentrated mainly on your **process**, rather than your product. For example, you find (through the use of metrics) programmer productivity is falling behind the norm and your source code contains more defects than acceptable. You may discover that if your developer increases the level of detail that goes into the design, your programmers will have an easier time translating the design into accurate code. This is a *process-focused* approach. If instead, you add another test cycle to catch the defects after the fact, you will be using a *"product-focused"* approach. Fixing defective products takes more time and money than building it right initially. ***Improving the development process always achieves lower costs and higher quality!***

Besides focusing on process, you need a systematic method to identify, correct, and prevent the root causes of problems. There are many approaches to describe this general procedure. One is the **Shewhart Cycle**, a systematic approach towards achieving uninterrupted quality improvement. Figure 15-1 illustrates this repetitive approach, the steps of which include:

- **PLAN** an approach for quality improvement. This involves studying the process flow and any existing data. Select possible improvements to the process, experiments to run, or data to collect.
- **DO** the planned activity. Implement the planned improvement effort. Train the people responsible for improvement implementation.
- **STUDY** the results. Measure the results of the implemented improvement effort. Analyze the data gathered.
- **ACT** on the results. If the effort was truly an improvement, standardize and document it. If it was not successful, determine how to improve it.
- **REPEAT**. Continue the cycle again by planning and carrying out further activity. [ESD94]

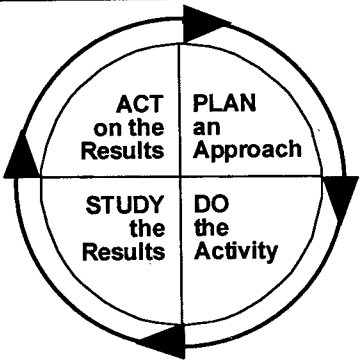


Figure 15-1 The Shewhart Cycle

The fundamental goals of software process improvement transcend the goals for all management activities, ***with quality as number one***. They are to:

- Increase software product quality and user satisfaction,
- Increase development productivity,
- Reduce development cost,
- Reduce schedule and technical risk,
- More accurately estimate costs, schedules, and resources, and
- Reduce scrap and rework.

Methods and techniques for process improvement abound. You must choose those that best suit your program, which might well be a custom-designed approach based on an assessment of your own development process. [KRASNER91]

In contrast to DoD software program failures, numerous studies have been conducted on the success of software development in **Japan**. With few exceptions, the Japanese are achieving very low software defect rates (approximately two orders of magnitude lower than the best American software companies). These studies cite data that confirm Japanese companies are developing custom software packages (similar to large DoD procurements) in ***35% less time than US companies do***. These lower defect rates and shorter schedules have translated into significant quality gains at lower cost. ***It is possible to produce higher quality software, cheaper, quicker, that is easier to maintain***. [BAKER92] US companies are finding the same is true. Quality doesn't cost; it pays! Many examples illustrate that increased costs associated with process improvement and adequate training are normally amortized within 12 months. Commitment to relentless process improvement with better business practices works for software as well as hardware.

To achieve process improvement, you must measure process effectiveness against stated objectives, such as lowering defect rates in delivered products. Measuring a process for the purpose of analysis-driven improvement includes indicators of **product** (quality) and **management** (estimating, planning, and monitoring) factors. **Process effectiveness** (and opportunities for improvement) are assessed through quantitatively measuring product, process, and management quality. Some relationships between these types of measures and the general goals of each are suggested in Figure 15-2.

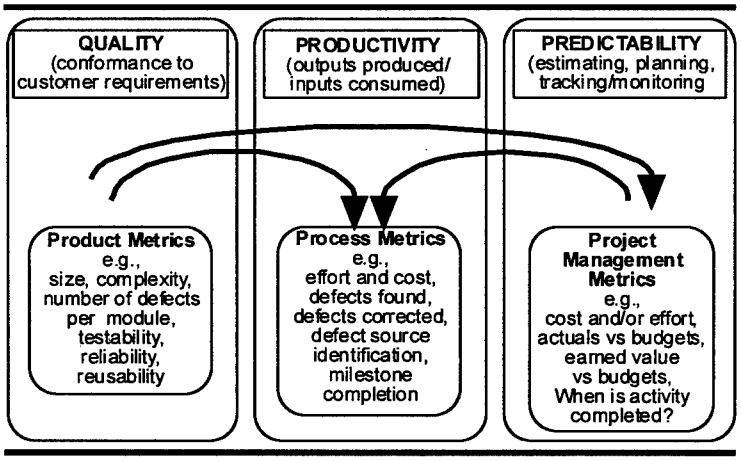


Figure 15-2 Process Measurement Metrics [STARS92]

CHAPTER 15 Managing Process Improvement

NOTE: See Chapter 7, *Software Development Maturity, "Capability Maturity Model (CMM℠)"* which is aimed at assessing *"process."* See Chapter 8, *Measurement and Metrics* for an in-depth discussion on how to set up a metrics program.

Data must be analyzed to identify weak aspects of the defined process and to provide feedback to improve the process. This can be a process used within the development program (*intra-development* process control or improvement) or within the framework of a database of process definitions, thereby leveraging process improvements across programs and organizations (*institutionalized organizational* or *inter-development* process improvement.) [STARS92]

The software problems identified in this book have affected software developments for decades. It is, therefore, important to recognize the benefits of analyzing past program problems. Applying the principles of TQM and the benefits gained from lessons-learned will help improve the development process. As your developer implements a structured approach for process improvement, they will be identifying problems, analyzing data, evaluating possible solutions, and working with teams. Placing emphasis on ***process***, rather than product, results in a successful, defined, refined process and product. Table 15-1 (below) outlines the Software Program Managers Network's **Flight Plan for Success** in managing large-scale software-intensive programs.

NOTE: The Air Force Process Improvement Guide provides an overview of a range of measurement tools you can use for process improvement and lists where to obtain additional information about them. [ESD94]

KEY AREA	KEY ACTIVITIES
The Basics	• Determine what staff you have to meet the program objectives • Determine the extent to which your program is sufficiently financed (including adequate risk reserve) requirements • Verify program requirements • Identify major stakeholders • Understand what your customer expects
Preparation	• Read and fully understand the nine Software Acquisition Principals, Best Practices, Caveats, and Targets *[see Chapter 2, DoD Software Acquisition Environment]* • Use the Little Yellow Book of Software Management Questions and the Project Breathalyzer *[see Chapter 16, The Challenge]* to determine program status
Risk	• Designate a "*Risk Officer*" • Identify top ten program risks *[see Chapter 6, Risk Management]* • Establish risk database • Prioritize all risks • Get high risks off the critical path where possible • Establish anonymous risk reporting mechanisms
Program Planning	• Ensure program planning is based on explicit requirements • Ensure plans are updated to reflect changes in program conditions and scaled to available resources
Metrics and Visibility	• Establish and follow a program-wide metrics plan • Establish a basic earned-value performance tracking management system • Monitor the critical path • Know your Cost Performance Index (CPI) and To-Complete Performance Index (TCPI) • Track employee overtime hours • Track rework metrics • Track productivity metrics • Conduct test coverage analysis • Conduct complexity analysis
Defect Tracking	• Track defects by severity and originating module • Track identified defects to their source (e.g., to the specific module, if in code) • Establish a process for collecting defect data • Use defect data to improve products and processes
Quality Gates and Inch-Pebbles	• Utilize binary quality gates at the inch-pebble level • Use quality gates to assess progress made against the program plan • Use quality gates to assure completion of low-level engineering tasks
Inspections and Reviews	• Conduct formal inspections (e.g., Fagan inspections) during all program phases/activities — ensure that the customer actively participates in them • Base all inspections on agreed-to, measurable criteria • Arrange frequent (but small scale) informal program reviews • Control all products used by or approved at a review

Table 15-1 Flight Plan for Success

KEY AREA	KEY ACTIVITIES
User Involvement	• Establish integrated product teams (IPTs) — multi-disciplined support team including users • Ensure that a "user's manual" for the system his been written, and approved by the user community • Have end users review user interface prototypes • Involve user in program reviews and other activities • Plan and conduct a joint user/developer program meeting to validate user interfaces
Interface Management	• Ensure that all external interfaces have been identified and are being tracked • Plan to test all external interfaces • Identify all key interfaces and external systems that interface with yours • Have a program-level interface database • Coordinate changes to interfaces with all affected stakeholders
Configuration Management	• Establish a configuration management process which controls all shared information • Establish a change control process that includes users
Program-wide Visibility	• Ensure that the program plan, progress, and risks are visible to the entire development/maintenance team • Keep stakeholders informed of program status • Foster a "*no cover-up*" culture

Table 15-1 Flight Plan for Success (cont.)

PROGRAM/CONTRACT MANAGEMENT

Software development is a very unique, complex management challenge, the approach for which must depend on your individual program needs. The process-approach to management has proven effective at the **Warner Robbins Air Logistics Center (WR-ALC)**, Robins AFB, Georgia, and is applicable to any software development organization (Government or industry). This approach is based on the application of sound, proven, process control techniques.

The first step in the WR-ALC approach is to ensure that those processes used to develop the software product(s) possess certain characteristics. These include:

- Software deliverable characteristics must be defined before product development begins;
- A software product that is workable, measurable, and deliverable must be produced at the completion of each process step; and
- Each delivered software product must satisfy the user's needs (or a designated portion thereof).

The next step is to put an **informal review process** in place, as well as a **deliverable repository**. The informal review process is implemented to determine if deliverables meet their intended design objectives. The repository is structured so it will *not*, under any circumstances, accept a deliverable unless, and until, the review activity agrees it properly meets its specification as defined in the **SDP** *[discussed in Chapter 14, Managing Software Development]*. It is only when the repository accepts the deliverable that the development organization may receive credit, and therefore, earn compensation for the work performed. Care must be taken to ensure that the software development process is small enough so it represents a complete task that provides a distinct deliverable. Likewise, the review process must not be so large that it is overwhelming, lethargic, or meaningless. The development organization must posture itself to insure that each process is a credit/value/revenue earning entity. **Earned-value** is tied to objectively observable steps, deliverables, and milestones, as illustrated in Figure 15-3. Each unit is either completed (or not completed. No credit is earned for incomplete units and value is only earned after completion of a milestone. Resources provided to the entity may ebb and flow as activities performed during the process vary. The assignment of process resources is one of the metrics reviewed as the deliverables are produced (or not produced).

An important step in this process-approach is **process evaluation**. As software deliverables are created and sent to the repository (a one-way flow), resources used to produce deliverable(s) must be tracked. **Resource tracking** involves measuring actual resource expenditures against SDP projected resources. At review cycle conclusion, action must be taken to address any problem area that caused the deliverable not to be produced as planned. Problems might include: inadequate planning, insufficient type/number of resources committed, or personnel not properly trained. If the deliverable was not developed as planned, the development process must be revised to insure it will work more efficiently in the future. [WEISS92]

In 1984, the earned-value approach was used by the **F-16C/D** SPO during the Block 25D cockpit avionics remechanization, the delivery of which was to contain the complete **Advanced Medium-Range Air-to-Air Missile (AMRAAM)** capability. When the software contractor failed to deliver the second OFP on schedule, the SPO had objective earned-value data describing the dollar value of the delinquent work. This provided a basis for ***withholding progress payments***. When the hardware contractor slipped delivery on the radar hardware (the APG-68), the Air Force withheld 10.6% on payments for deliveries

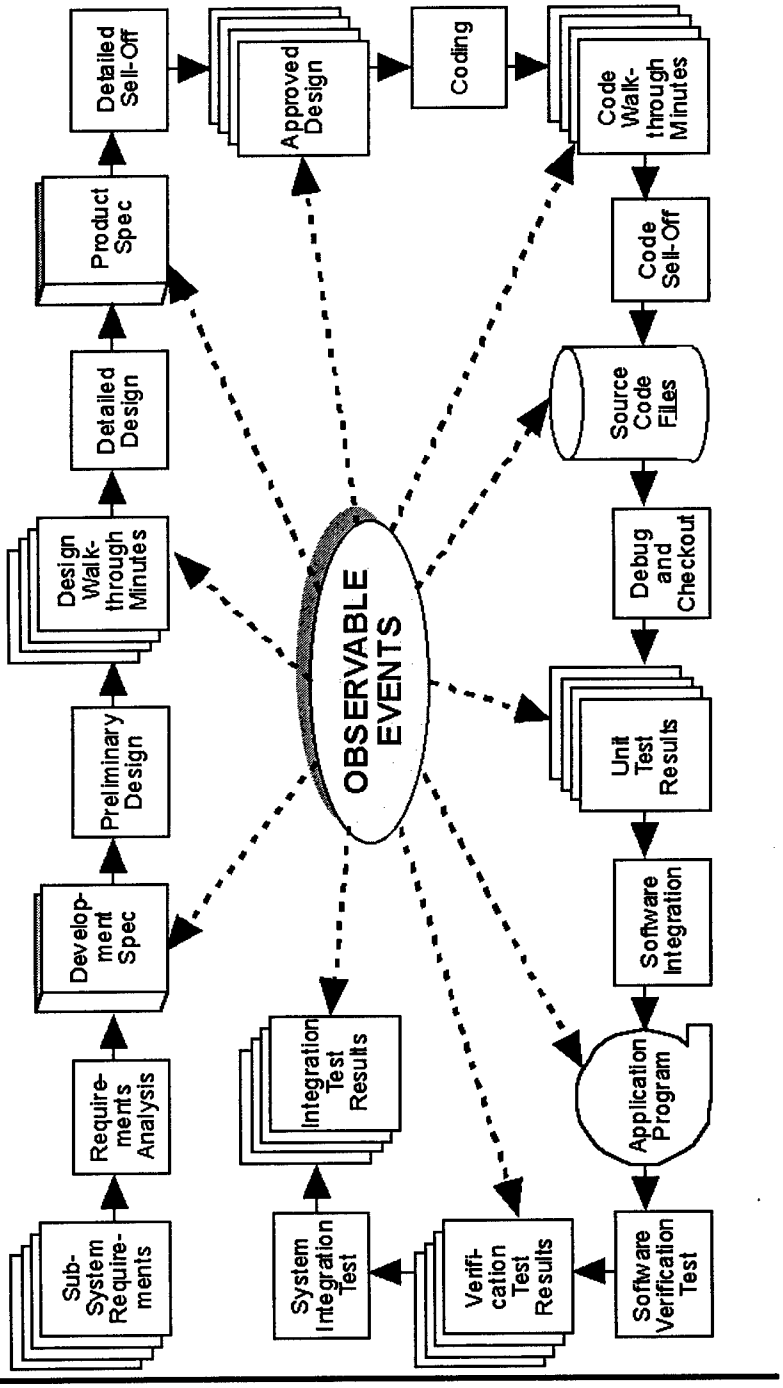


Figure 15-3 Earned-Value through Objectively Observable Milestones

with less than full specification compliance. By the end of 1984, approximately $24.7 million had been withheld from the software development contract and $21.8 million from the hardware production contract. By mid 1985, a total of $93 million in progress payments had been withheld.

Use of the earned-value approach was an effective way to get these contractors' attention. A hefty message was sent where it hurt the most — in their pocketbooks! Early in 1984, the software contractor quickly informed the SPO of the process improvement activities they were implementing to focus attention and manpower on solving their software development problems. These included:

- They withdrew from competition on several other electronics contracts to free up experienced people for assignment to their avionics efforts.
- They reorganized and reduced the supervision ratio from 40:1 to 20:1, allowing first-line supervisors to take a more active role in technical direction and group planning.
- They split their software engineering/development section into three sections: F-16 Systems Engineering, F-111 Systems Engineering, and Validation and Testing.
- They established a Planning and Control Group which studied the feasibility and impacts of taking on new corporate commitments and the manpower needed to support them. The group also tracked programs, alerted management to schedule deviations, and developed recovery plans.
- They consolidated testing activities and talent into a centralized location.
- They set up an Executive Software Review Committee to assess how the company developed software, to recommend improvements, and to implement long-term solutions for reducing development schedules. A single point of contact was appointed who reported directly to their vice president for research and engineering and to the F-16 Program Director.
- They increased management commitment to place greater emphasis on software development and began an accelerated manpower ramp up in their avionics department to accommodate future block schedules.

Process improvement is always more successful when a government/ contractor team effort. The SPO sought to improve their requirements process by better defining the remechanization task. Line pilots reinforced previous test perceptions that routine tasks were, in fact,

more difficult in the F-16C/D than its predecessor. They formed a General Officers panel that flew orientation flights on the new system. An independent Cockpit Review Team was also implemented that used simulators and developmental aircraft to check and refine the **Pilot Vehicle Interface (PVI)**.

Two years after overcoming their difficulties, the dual team watched the first Block 25B roll off the production line. Over the next several years, the F-16 was equipped with even more capable software. With the implementation of management process improvement and the appropriate use of metrics tools to estimate software development workload (e.g., COCOMO, as discussed in Chapter 8, *Measurement and Metrics*) and track intermediate step completion (e.g., earned-value), the F-16 software upgrade went as smoothly as planned. The F-16's revolutionary software systems were integral to its success (and the success of other Air Force weapons systems) in neutralizing the enemy during the **Gulf War**. Whether the requirement was for precise bombing against fixed or mobile targets, carrying out *"killer scout"* missions, identifying and marking elusive ground targets for other attack aircraft, or providing reinforcement to air-to-air-combat missions, the F-16 proved to be a formidable adversary and an awesome team player in history's greatest air campaign. The successful upgrades to its software-controlled systems were key to the F-16's multi-tactical role in achieving an unprecedented victory for American air power.

Cost/Schedule Control System Criteria (C/SCSC)

A **C/SCSC** and a **Cost Performance Report (CPR)** should be employed on major contracts with an RDT&E budget greater than $70 million and a procurement value of more than $300 million (in FY96 constant dollars). The C/SCSC system was designed to produce a single database of management metrics for all major DoD acquisitions. *[Note: C/SCSC, or a commercial equivalent, should also be considered for high risk MIS programs.]* Cost data are provided through the standard CPR so managers (both Government and industry) can determine cost and schedule performance using earned-value techniques. Earned-value output data from these systems are then used to make independent assessments of contract cost and schedule status. As work proceeds against a budgeted program, dollars are *earned* as they are expended. Earned-value is a measure of progress against the plan. [HEWITT93]

To ensure uniformity across DoD programs, the contractor's management system is validated against a set of predefined C/SCSC criteria. Contractors are not required to revise existing systems, except as necessary to satisfy DoD cost monitoring requirements with accurate data. The documentation and data required are limited to the minimum needed to satisfy requirements. The contractor's C/SCSC must produce data that:

- Indicate work progress,
- Relate cost, schedule, and technical accomplishment,
- Are valid, timely, and auditable, and
- Provide DoD managers with practical summaries.

The important consideration is that financial management be integrated with program management. The contractor should breakdown all known work for the next six months into detailed **work packages** using the contract WBS supplied on contract award. A monthly **contract budget** is then developed based on the work package start and stop dates and on the detailed budget for each package. [MARCINIAK90] The budget includes everything necessary to complete the work package, such as overhead, materials, labor, and schedule. This is a crucial step as it establishes the **budget baseline** for the entire contract.

As the work proceeds, expenditure and progress reports are used to track each work package. Performance is measured by comparing three quantities: **budgeted cost of work performed (BCWP)**, **actual cost of work performed (ACWP)**, and **budgeted cost of work scheduled (BCWS)**. BCWP is the baseline measure of earned-value against the overall plan for the contract. Following each reporting period, BCWP is compared to ACWP for each work package to determine **cost performance** (i.e., is the actual cost greater or less than the budgeted cost for each package). BCWP is also compared to BCWS to determine **schedule performance** (i.e., is each package ahead or behind schedule). Figure 15-4 is a simple illustration of how you can interpret these figures. In this example, BCWS exceeds BCWP. This negative variance implies the program is behind schedule. Likewise, the ACWP is greater than the BCWP, indicating that the program is headed for a cost overrun. The contractor is often required to report to the Government when cost or schedule variances exceed thresholds established by the contract.

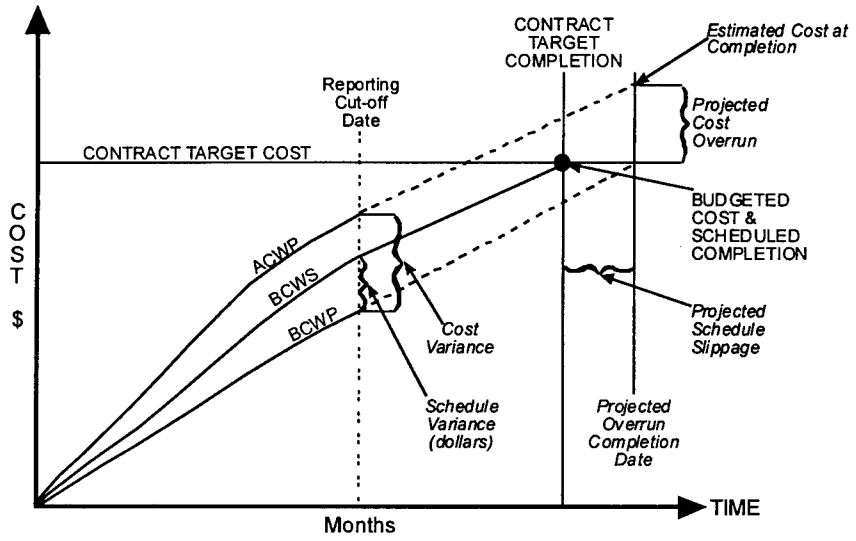


Figure 15-4 C/SCSC Earned-Value Analysis

CAUTION! It is important for your contractor to understand the causes of variances so they can take appropriate remedial action. It is also important for the Government to understand the causes of variances that will impact final program cost or schedule. Do not, however, set the reporting thresholds so low that your contractor spends all their time writing variance analyses rather than developing and delivering your product!

In 1991, the Office of the Under Secretary of Defense (Acquisition) sponsored a government/industry **process action team (PAT)** study of current RFPs and major contracts to determine why deficiencies in the C/SCSC system were occurring. They found that major problems, emanating from micro-management on the part of DoD and improper reporting on the part of contractors, indicate a need to streamline the system to reduce cost monitoring waste. The problems they encountered were:

- WBS problems *(e.g., levels too low; functional elements; or color of money)*,
- C/SCSC implementation problems *[e.g., C/SCSC is inappropriately required; on non-major contracts C/SCSC, CPR, and Cost/Schedule Status Report (C/SSR) are all required; improper subcontract flowdown; or a pert cost type system is specified]*,

- Inadequate SOWs, and
- Reporting problems *[e.g., excessive CPR variance analysis; variance analysis guidance omitted; CCDR plans/reports; level-of-effort exceeded given percent required for notifying primary contracting office; or ANSI-X12 was not specified for* ***Electronic Data Interchange (EDI)****].*

Another common problem occurred when unrealistic baselines were established. While contractor cost management systems were closely aligned with C/SCSC compliance, their baselines could not be reset without contract modification. Large variances (attributable to the unrealistic baseline) occurred between baselines established early in the contract period and actual performance. To compensate, internal *ad hoc* systems began to evolve which documented performance against modified baseline work plans. This created a situation where the contractor *planed* activities using their *ad hoc* plan, and reported progress against a baseline they no longer used. The major symptom of these practices was ***poor baseline integrity***. Baseline integrity problems include:

- **Front-loaded baselines** delay the visibility of contract cost problems, as illustrated in Figure 15-5;

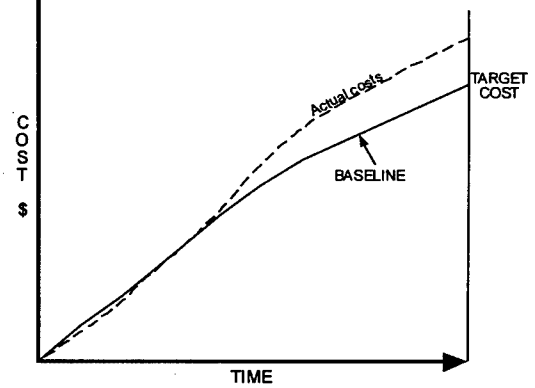


Figure 15-5 Front-loaded Baseline

- **Rubber baselines** also delay the visibility of contract cost problems since they are revised midstream in an attempt to match actual costs while staying within budgeted costs, as illustrated in Figure 15-6;

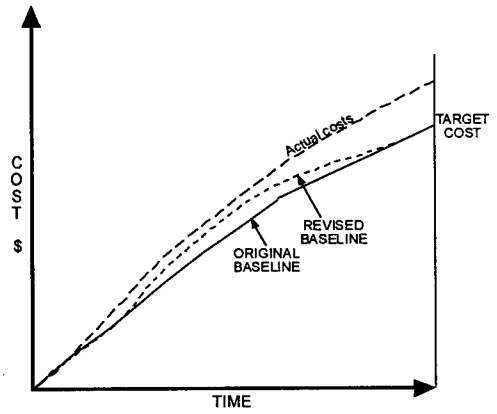


Figure 15-6 Rubber Baseline

- **Internal replanning** causes large amounts of undistributed budget and excessive use of summary level planning packages, as illustrated in Figure 15-7;

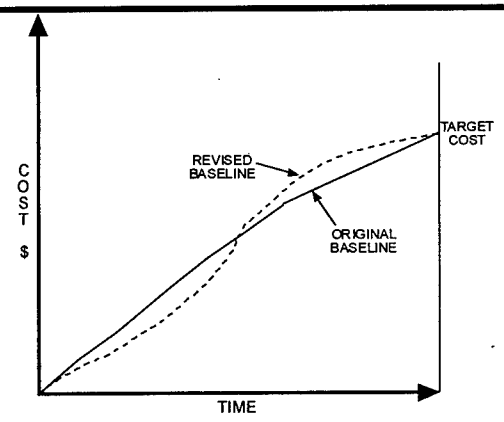


Figure 15-7 Effects of Internal Planning

- **Baseline budget exceeds contract value** which builds overruns into plan and cost reports that do not depict true contact status (only performance against an *ad hoc* plan), as illustrated in Figure 15-8 (below); and

- **Erratic variance** patterns.

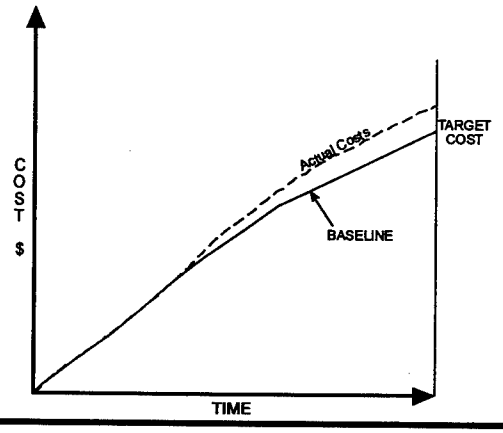


Figure 15-8 Baseline Budget Exceeds Contract

The PAT team concluded that baseline integrity is a function of data reliability, and problems associated with C/SCSC reviews were caused by poor contract management. Because the program management office owns the contract baseline, program management participation in the review process is key. The C/SCSC must not be viewed as merely a checklist drill, but as a program management tool. Your government team must, therefore, be prepared through pre-review analysis and training. The baseline must always be assessed before or during the C/SCSC review. Your technical staff must examine baseline content and time phasing so they understand the source of performance data, thus improving contract management understanding. Too often, earned-value insights remain the sole province of government/contractor cost analysis support staffs. Problems do not surface to the decision maker level until it is too late for remedial action. Therefore, ***earned-value must be a integral part of your "integrated" software development team approach***. When management staffs are held accountable for earned-value analyses, they begin to understand earned-value implications. [OSD/ A93]

Cost performance problems often surface as **estimate-at-completion (EAC)** discrepancies. EAC is computed based on actuals to date, plus an estimate of future costs based on the contractor's ***demonstrated cost*** and ***schedule efficiency***. The **schedule performance index (SPI)** for efficiency is:

$$SPI_E = \frac{BCWP}{BCWS}$$

The **cost performance index (CPI)** for efficiency is:

$$CPI_E = \frac{BCWP}{ACWP}$$

Values greater than 1.0 represent performance better than planned (more efficient), and values less than 1.0 represent less efficient performance. Schedule inefficiencies, as well as cost inefficiencies, can contribute to cost overruns at completion. If your development effort is more than 15% complete and you are overrunning your baseline estimates — your percent of overrun at completion will be ***greater*** than your percent of overrun to date. This prediction is based on more than 700 DoD contracts since 1977. The conclusion is that, ***you cannot recover!*** If you have underestimated in the near-term, there is little hope you did much better on your far-term estimates. The solution to this problem is not to worry about recovering, but to figure out how you can keep from getting worse. Thus, you must adjust your far-term estimates. You must assume your future work will overrun at the same rate as your current behind-schedule condition, and adjust your far-term projections to this rate.

NOTE: Consult with your cost analysis support staff if you need help in computing an EAC for your contract.

Earned-Value Software Metrics

According to Christensen and Ferens, there are several software metrics appropriate for BCWS, BCWP and ACWP. To be useful, a metric should be:

- **Relevant** to the work being measured,
- **Explicit** (directly measurable),
- **Objective,**
- **Absolute** (able to be assessed without reference to an average),
- **Timely** (available early in the program), and
- **Independent** from the influence of personnel working on the program.

Of these, relevance is the most important property for earned-value. The first two metrics are also appropriate for earned-value measurement — with ***objectivity*** the most appropriate for ACWP. The remaining four metrics are more useful in investigating variances than in the direct measurement of earned-value or actual costs. The following describes each metric and its relevance to the earned-value approach.

NOTE: See Chapter 8, *Measurement and Metrics*, for an in-depth discussion on metrics.

- **Requirements and design progress.** This metric is based on the number of CSCI requirements determined during the first two phases of software development. The requirements are detailed in several documents [**(System/Segment Design Document (SSDD), Software Requirements Specification (SRS), Software Design Document (SDD)**] written during these phases. As illustrated in Figure 15-9, the planned and actual CSCI requirements are used for determining BCWS and BCWP, respectively. Figure 15-9 also illustrates that the total CSCI requirements may change. In addition, counting the requirements can be difficult. If these limitations can be overcome, this metric is a viable tool for earned-value application, especially early in the program.

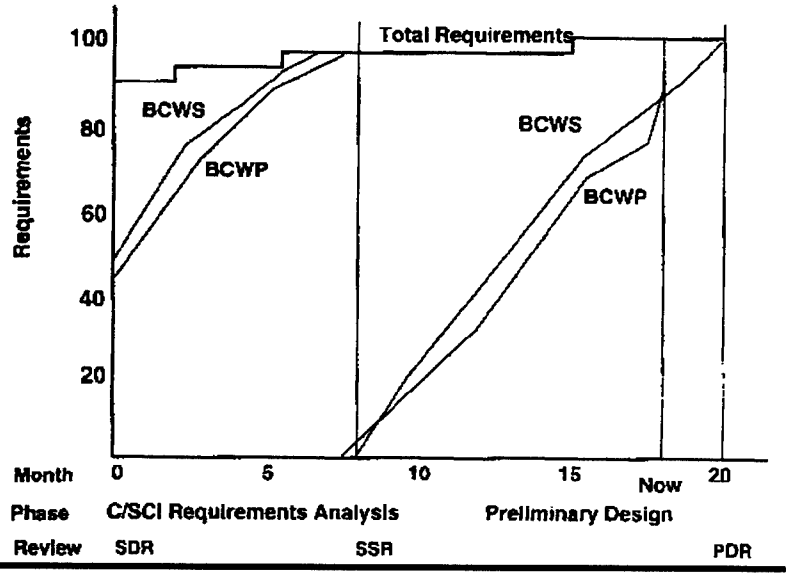


Figure 15-9 Requirements and Design Process Metric

- **Code and testing progress.** This metric is based on the number of CSUs that have been designed, coded, and tested. As illustrated in Figure 15-10, it is appropriate after the second phase of software development. Like the previous metric, the planned and actual CSUs represent BCWS and BCWP. In addition, the total number

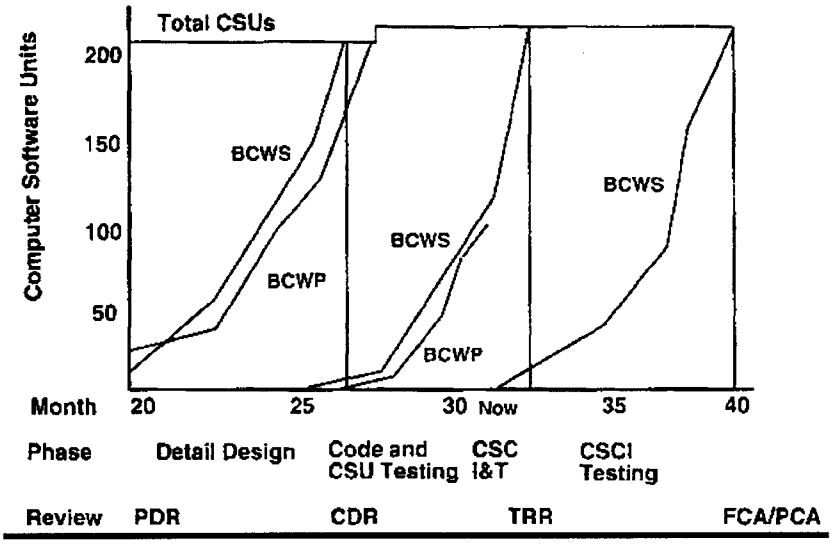


Figure 15-10 Code and Test Progress Metric

of planned CSUs for each phase represents the end point of the performance measurement baseline for that phase. Generally, this metric is easier to measure than the previous one. CSU progress can be measured using a unit development folder or similar technique. Also, more detailed information is known about the software program in these later phases.

- **Manmonths of effort.** As illustrated in Figure 15-11 (below), this metric is based on manmonths throughout the program. As such, it is particularly useful for measuring ACWP because the costs of software development are almost entirely labor-related. Using planned manmonths (person-months) for BCWS and BCWP is inappropriate because available estimation methods may be inaccurate, and the time spent on the program may not correlate to progress. Nevertheless, this metric is useful, if only because it is the single metric in this collection that directly reflects ACWP.
- **Software size.** This metric tracks the size of the software during the entire program. Usually, size is expressed in source lines-of-code (SLOC) or function points. The total size may be divided into categories of new, modified, and reused code. Since there is a direct relationship between size and effort required, this metric is helpful in estimating actual cost. However, effort required and actual progress may not correlate; accordingly, the method may be inadequate as an earned-value metric, and should be used as a technical parameter to investigate the cause of cost variances based on the other metrics.

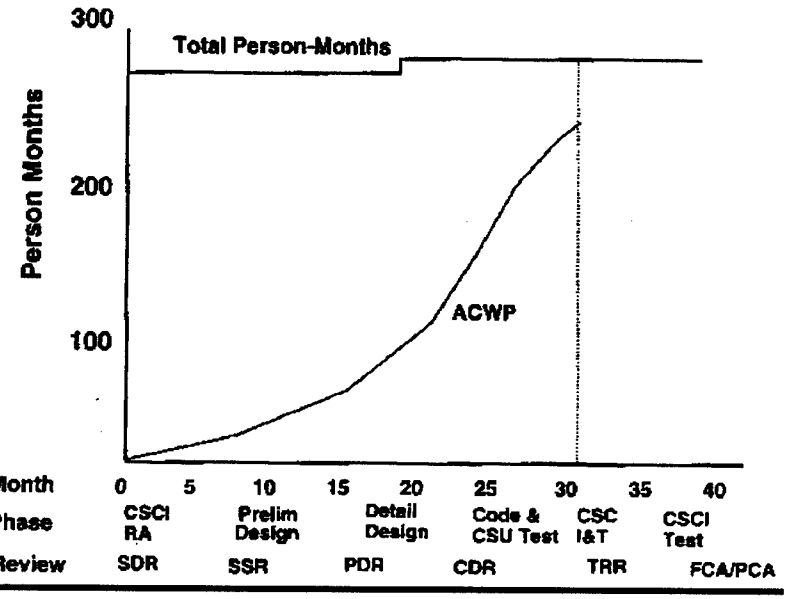


Figure 15-11 Manmonths Progress Metric

- **Computer resource utilization.** This metric is a measure of the available computer hardware timing, memory, and input/output (I/O) resources consumed by the software. It is closely related to the software size metric in that increases in total size result in a greater percentage of hardware resources utilized. Like software size, this metric is helpful early in the program for determining the causes of variances.
- **Requirements stability.** This metric has similarities to the requirements and design progress metric. Like that metric, requirements stability tracks total requirements; however, it also tracks the number of changes (additions, deletions, and modifications) made to requirements throughout the entire development process. Numerous or frequent changes result in additional effort required, and explain unfavorable cost and schedule variances.
- **Design stability.** This metric is like requirements stability in that it tracks the number of changes to the detailed design (CSUs). Like the code and testing progress metric, it is useful later in the program, after preliminary design is complete. Frequent lower-level design changes result in additional effort required.

NOTE: See Addendum A of this chapter, *"Improving Software Economics in the Aerospace and Defense Industry."*

Table 15-2 lists the seven metrics just discussed, and indicates the role each metric can play in an earned-value performance measurement system. The table also indicates how well the metric satisfies the seven desirable properties of software metrics. Because these properties are nearly identical to the goals for earned-value described in the C/SCSC, they are viable candidates for earned-value application, especially the first three listed in the table. Granted, the metrics described here are not the only ones. You should also consider the **quality metrics** *[discussed in Chapter 8, Measurement and Metrics]* that track defects, complexity and modularity. While these metrics do not directly relate to earned-value measurement, they help measure quality, which is the *sine qua non* of software programs. Using them in tandem with the ones recommended for earned-value is highly recommended. [CHRISTENSEN95]

METRIC	DESIRABLE ATTRIBUTES OF SOFTWARE METRICS						ROLE IN EARNED-VALUE	PHASE(S) IN SOFTWRE DEVELOP MENT PROCESS
	RELE-VANT	EXPLICIT	OBJECTIVE	ABSOLUTE	TIMELY	INDEPEN-DENT		
Requirements and Design	Yes	Yes	Somewhat	Yes	Yes	Somewhat	BCWS and BCWP	CSCI Requireme ts Analysis and Architectur Design
Code and Test Progress	Yes	Yes	Somewhat	Yes	No	Somewhat	BCWS and BCWP	Detailed Design, Code and Unit Testing, CSC Integration and Testing and CSCI Testing
Manmonths Progress	Yes	Somewhat	Yes	Yes	Yes	Somewhat	ACWP	All phases
Software Size	Yes	Somewhat	Yes	Yes	Yes	Somewhat	Technical indicator for variance analysis	All phases
Computer Resource Utilization	Yes	No	Somewhat	Yes	Somew hat	Yes	Technical indicator for variance analysis	All phases
Requirements Stability	Yes	Yes	Yes	Somewhat	Yes	Yes	Technical indicator for variance analysis	All phases
Design Stability	Yes	Yes	Somewhat	Somewhat	No	Yes	Technical indicator for variance analysis	All phases

Table 15-2 Software Metrics for Earned-Value

Lessons-Learned from the Navy Seawolf Program

NOTE: See Chapter 5, *Ada: The Enabling Technology*, for a brief description of this program. Also see Volume 2, Appendix O, Chapter 15 Addendum C, "*Lessons-Learned from BSY-2's Trenches.*"

- Establish direct lines of communication among program office, developers, and IV&V for identifying and resolving problems.
- Exchange information among stakeholder agencies for increased visibility program-wide.
- Improve prime contractor control through weekly monitoring and quarterly audits of subcontractor efforts. Require mandatory attendance at technical and working group meetings for all team members.
- Beginning early in the program, routinely review processes related to software development (e.g., CM, SQA, testing).
- Establish a streamlined waiver request process for reporting contract deviations.
- Develop an extensive Ada training program (with concentration on software engineering) that is customized for application-specific requirements.
- Involve government representatives from multiple disciplines (quality, test, readiness, operability) in the review of contractor processes.

Lessons-Learned from SSC and CSC

NOTE: See Chapter 5, *Ada: The Enabling Technology*, for a description of the programs upon which these lessons-learned are based.

- It is essential that management structure and communications procedures allow for the diplomatic and timely resolution of development problems. Use team building techniques to co-locate government/contractor teams and cultivate open lines of communications.
- Development of a pre-review checklist ensures all necessary actions (e.g., arranging for the facility, notification, documentation preparation, distribution, etc.) are accomplished.

- Lessons-learned must be documented as they occur because they are not easy to recall during the heavy workload associated with preparing for a major review. A standard form should be designed to aid in the collection of input from team members. Each task should appoint a POC to collect and document task input on a continuous basis.
- Make sure the metrics used agree with the development methodology; e.g., the SDP should define a set of metrics tailored for use with object-oriented methodologies.
- *"Kits"* are informal internal communications tools. Any design decision, regardless of how specific it is, should be documented in Kit format. If that decision is modified, then that Kit should be revised with the new information.
- An integrated toolset enhances productivity. A tool that allows the use of any word processor, without losing the ability to directly port to other development and configuration management tools, saves valuable manhours spent retyping and reformatting documents.
- Previous lessons-learned must be heeded. A single point of contact should be established for coordination and dissemination of lessons-learned.
- To maximize the use of in-house expertise in problem solving, publish a list of experts and their corresponding areas of expertise.
- Interfaces are key to establishing the system's classification and security needs. Late identification of these key factors will adversely impact the program's schedule. Planning for external interfaces must begin in the Task-Level Planning Phase with interface requirements gathering, identification, and agreements.
- A system must operate effectively and support the needs of the environment in which it is placed. A *business analysis* establishes the policies and practices of the system's users, and ensures that the requirements gathered and the system developed will meet users' needs.
- The process for learning how to use all the reuse repositories is time consuming because each is different. When possible, each team should assign reuse repository responsibilities to a single individual or small group of individuals.
- To establish, refine, and maintain a repeatable process, a lessons-learned process improvement form should be established. The form provides a way to capture the results of an activity.
- To facilitate the smooth release of a document which is revised periodically (such as the SDP), it is necessary that the changes be incessantly gathered and inserted into a master copy of the document rather than waiting until near delivery date to do it all.

- Because DoD does not train its officers to be dedicated software program managers, the situation often occurs where program managers find themselves inadequately trained for the job at hand. The benefit of a proven and repeatable software development process is that a checklist of key practices required for proper software development may be used with confidence, even by someone inexperienced in the processes involved.

SOFTWARE QUALITY ASSURANCE

Offerors' proposals should discuss their corporate program for **software quality assurance (SQA)** to include: organization, procedures, and personnel. Be aware, ***the traditional review-oriented approach to software quality assurance often fails to achieve quality software.*** A quality assurance approach emphasizing the evaluation of *completed* (or close to completion) software yields very little towards the goal of quality software. This type approach is a costly way to detect and correct defects already built into the code. It does not reduce or prevent the occurrence of errors. Software quality cannot be inspected or tested in, it must be ***built-in***. A SQA program must emphasize early SQA involvement throughout the software development process. To fully integrate SQA into your software development, you should require that SQA status is reported monthly at all program management and software reviews. *[MIL-STD-498, Appendix D, identifies those software products that should undergo quality evaluations. For an example of SQA efforts on the F-22 Program, see Volume 2, Appendix K, Software Support.]*

In source selection, you must look for the offeror's approach to SQA that not only conforms to specifications, plans, and procedures, but concentrates on a quality process for **error prevention**. [BAKER92] You should also use these data to determine the level, extent, and type of **independent verification and validation (IV&V)** required for their software development effort. Obviously, the offeror with the most comprehensive, institutionalized quality control process will require the least IV&V. *[Quality is discussed in Chapter 9, Measurement and Metrics. A sample paragraph for including software quality assurance in the RFP is provided in Volume 2, Appendix M.]*

There is a distinction between quality control and quality assurance. **Quality control** is implementation (design/code/test)-oriented; it is the developer's task to ***build*** a quality product. **Quality assurance** is inspection-oriented; it is the **software quality assurance (SQA)** team's task to ***assure*** a quality product and ***process***. [GLASS92]

SQA must be an integral part of the software development effort. One approach is to have an independent team of people (who are not necessarily developers) review the development process. The SQA team's role is to monitor and verify that methods and standards are being properly implemented by developers. SQA, a discipline in its own right, should consist of quality experts who establish a strong quality program. [HUMPHREY90] A SQA program is:

> *A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements.* [ANSI/IEEE83]

The SQA can take the form of a quality assurance team, a change/ inspection review board, a product test group, independent verification and validation personnel, or a combination of these. [GLASS92] The SQA element should have both government and industry team members with knowledge of statistical methods, quality control principles, the software development process, and an ability to deal effectively with people in a constructive manner. [HUMPHREY90] Figure15-12 (below) illustrates how SQA is integrated into the entire system life cycle.

Another important quality team, the **process action team (PAT)**, is made up of small groups of developers and software verifiers who analyze defects and identify their causes. Team members are the analysts, programmers, and verifiers for whom software product quality is their daily responsibility. These teams also determine how to remove defect cause, and subsequently, implement process changes. The idea is to have those who execute the development process also execute the improvement process. [KINDL92]

The effectiveness of SQA and process action teams is rooted in the strength of your management and of their combined contribution to a common quality goal. Once these teams are formed, there is a danger that the responsibility for quality itself can become split between the developers and the quality experts — thus diluted. It is your responsibility to make sure this does not happen. A clear definition of individual team responsibilities is vital. How these teams work together, from an overall program management perspective, is your call. Whichever method you choose, it is vital to maintain organizational independence between the two. Figure 15-13 (below) illustrates how a development team might be organized to insure objectivity in a quality software process.

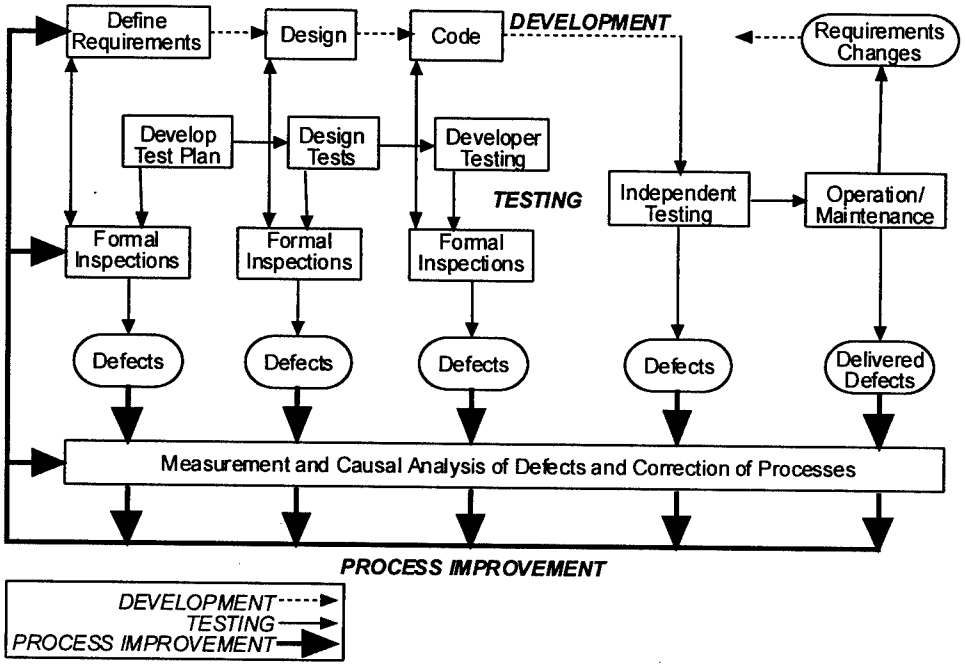


Figure 15-12 Software Quality Assurance and System Life Cycle [KINDL92]

Key to a successful quality program is total support and commitment from management. The responsibility to analyze and execute rests with the SQA and the developer. The responsibility to allocate resources and make decisions that support process improvement rests with you. [KINDL92]

As you learned in Chapter 14, *Managing Software Development*, engineering quality into DoD software requires that your developer inspect, test, and remove defects from requirements, design, documentation, code, test plans, and tests. Effective quality assurance means that standard procedures are established to measure defects, determine their root causes, and take action to prevent future occurrences. Built-in, as part of the development process, ***must be a procedure to change the process***, facilitating perpetual improvement. Such a process is self-correcting where future measurement provide convincing evidence of cost-effectiveness. Figure 15-14 illustrates the management factors that must be considered during the development process improvement.

CHAPTER 15 Managing Process Improvement

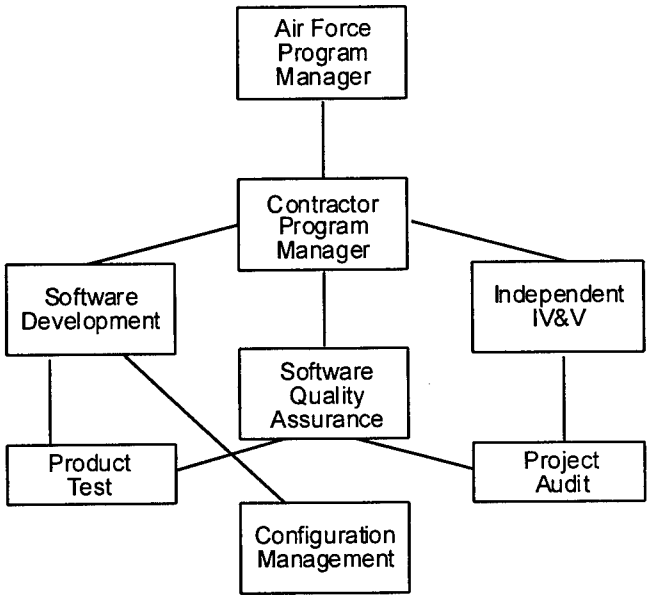


Figure 15-13 Sample Management Structure for an Independent SQA Element [GLASS92]

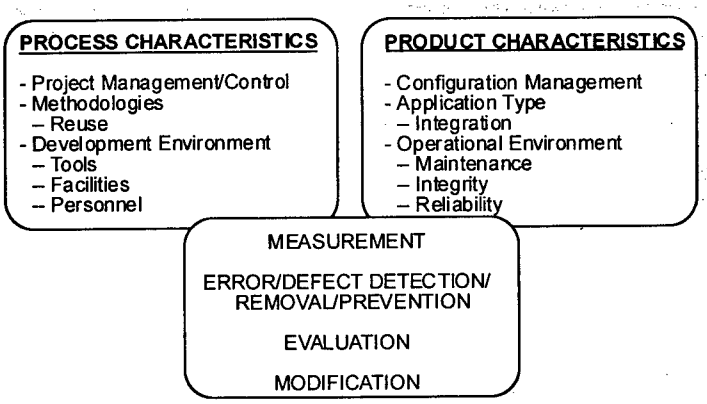


Figure 15-14 Management Factors to Address for a Quality Product [KINDL92]

Defect Prevention

A formal **defect prevention** program should be established that empowers developers and software testers to analyze the causes of defects, and to enact improvements to their own local development processes. This empowerment helps prevent future defect insertion and enhances the detection process. [KINDL92] Defect prevention directly relates to the quality of the development process. ***The degree of prevention is dependent on the degree of process improvement accomplished throughout the development.*** How a process improvement approach pays off with defect prevention is best described through example. Prior to implementing a process improvement approach, the design manager for the Boeing B-1B avionics program admitted, *"It used to take about 25% more effort to correct all the design errors than it took to complete the original software design."* [BENDER90] Prior to starting work on the newly awarded B-1B **Short-Range Missile II (SRAMII)** integration contract, the Boeing software designers and engineers took an in-depth look at their previous software development process. They found that 25% of the effort was spent designing software, 6% on coding, and 29% on testing. The remaining 40% was spent fixing problems that could have been avoided with better upfront design. They concluded that, *"If we could do things better upfront, we could avoid the costs involved with inspection and testing, and we wouldn't have to fool around with changes and fixes later down the road."* [BENDER90]

A 20-member **process action team** met twice a week to eliminate the prior dependence on testing and to break down communication barriers. Detailed agendas of each bi-weekly meeting were distributed to ensure all stakeholders in the software effort attended. One of the first initiatives was to adopt a ***measure-of-quality*** (the number of defects per lines-of-code), and to track progress. After implementing improvements to prior software block developments, the frequency of defects during the SRAMII block development went down 20%. In addition, the contract schedule was met and costs were reduced by 60%. [KEOHLER90] The program manager recalled, *"The most important thing we learned was that the earlier we isolated the problems, the easier and less costly the fix."* [BENDER90]

NOTE: See Chapter 7, *Software Development Maturity*, "Benefits of Moving Up the Maturity Scale," for discussions on defect removal and prevention returns on investment.

Defect Causal Analysis

Perhaps the most important aspect of software process improvement is **defect causal analysis**. Quality software demands that defects be eliminated and prevented, not only from all products delivered to users, but also from all the processes that create those products. [HOLDEN92] As discussed throughout these Guidelines, finding and correcting mistakes makes up an inordinately large portion of total software development cost. ***The level of effort for rework to correct defects is typically 40% to 50% of total software development***, as illustrated in Figure 15-15.

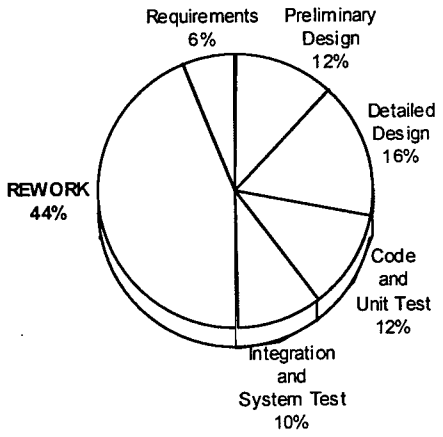


Figure 15-15 Defect Rework Hidden Cost [BOEHM89]

BE AWARE! Because this high level of rework does not reflect positively on developers, it is often not openly reported.

In addition, rework is distributed throughout the development phase as shown on Figure15-16 (below). The effort required to correct defects becomes compounded the later they are detected. The snowball effect on cost occurs because all the work completed after their insertion often must be reworked, as it was based on erroneous foundations. [BRYKCZYNSKI93[1]]

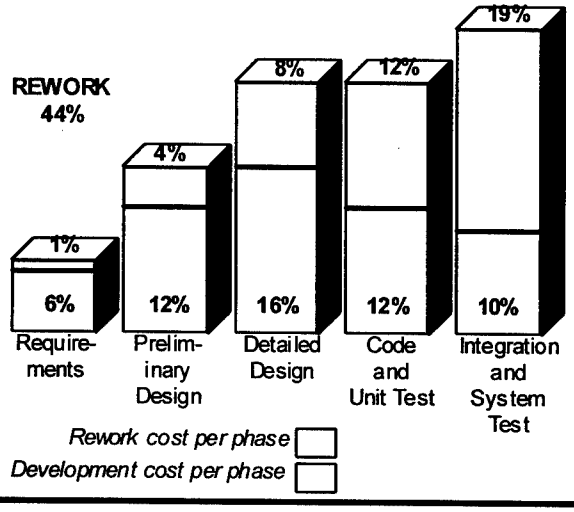


Figure 15-16 Rework Cost per Development Phase

Defects happen! If we cannot build defect free software, the cheapest, quickest, next best way to build good software is to learn from our mistakes and build it right the next time. **Defect causal analysis** is an effective method for improving software quality. It is an orderly technique for identifying problems and ***preventing defects***. Causal analysis focuses not only on defect discovery, but also on what caused the defect to occur. The importance of this SQA approach is that it provides the necessary feedback for the improvement of tool use, software engineering training and education, and ultimately, the development process itself.

Figure 15-17 illustrates the causal analysis process where, at the start of each development phase, an *entrance conference* is held to review deliverables from the previous phase, to review process methodology guidelines and standards, and to set the team quality goals. These exercises start the **defect prevention process**, since they concentrate the team's attention on the process-oriented development details soon to be performed. Throughout each phase, government audits, reviews, and formal peer inspections are scheduled, in addition to walkthroughs by corporate management. During these reviews, each work-product is checked for defects and rework is performed (as required) with follow-up reviews.

The most important activity during this process is the preliminary causal analysis of the defects themselves. During causal analysis meetings, individual problems are isolated and analyzed (somewhere

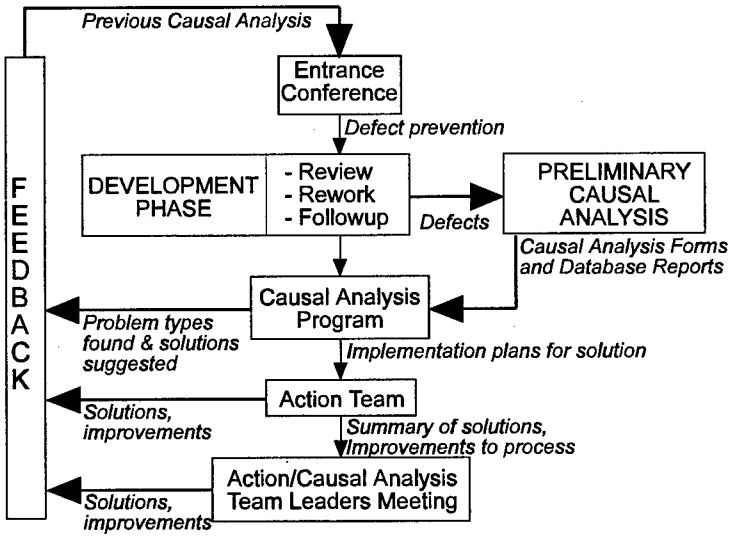


Figure 15-17 Defect Causal Analysis Process [YOURDON92]

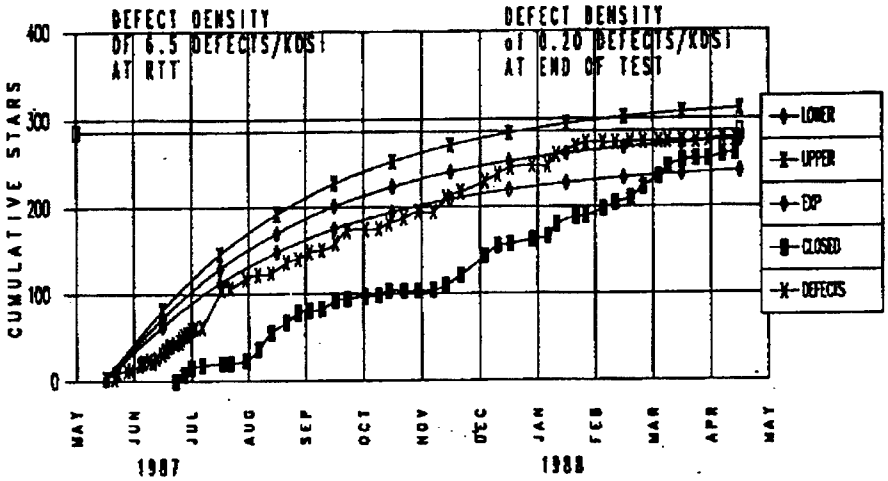
between 1-20 defects can be scrutinized). Each defect item is then added to a database where it is tracked by its description, how it was resolved, and the preliminary analysis of its cause. Another meeting is held at the end of the development phase which consists of a brainstorming session. The purpose of this meeting is to analyze defect causes, evaluate results versus goals set at the start of the phase, and to develop a list of suggested process improvements. The **process action team** (e.g., people from the tools, training, development, support, and testing group) must then respond to these suggestions. The process action team is responsible for:

- Action item prioritization,
- Action item status tracking,
- Action item implementation,
- Dissemination of feedback,
- Causal analysis database administration,
- Analysis of defects for generic classification, and
- Success story visibility. [YOURDON92]

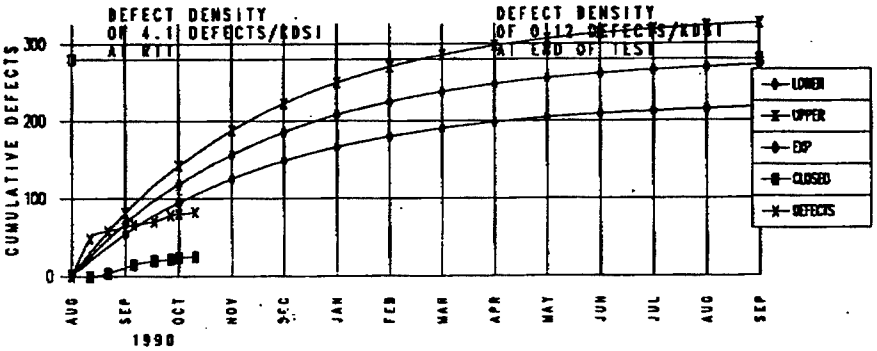
An example of a causal analysis tool is the **Software Defect Detection Model** used on the **F-16C/D** avionics program. The model is used to predict the total number of defects in core avionics operational flight programs (OFPs) and the detection rate for finding those problems. Use of the model has greatly reduced software defect

insertion over the span of F-16C/D avionics programs. Figure 15-18 illustrates how the defect insertion rate progressively decreased by 1/3rd between production Blocks 30 and 40 and again between Blocks 40 and 50.

(Block 30B Software Defect Density = 9.8 Defects/thousand delivered source instructions (KDSIs))



Block 40B Software Defect Density = 1/3 reduction in software defect insertion rate from Block 30B



Block 50B Software Defect Density = 1/3 reduction in software defect insertion rate from Block 40B

Figure 15-18 F-16 Software Defect Detection Model Results

The F-16 model is driven by discovered defects which are logged into a simple database that categorizes them as open, closed, or canceled. An analysis of each defect is included, in addition to whether a fix was identified and confirmed. Defects causal analysis gives visibility into the quality of the test program, and maximizes the cost effectiveness and use of test assets. This defect measuring and correction process has been successful in transferring defect discovery to the upfront development phases, and away from defect detection during operational flight testing where human life is at risk.

Defect Removal Efficiency

An anomaly about defect detection in large software-intensive systems was identified in studies performed by major corporations (IBM, DEC, Wang, AT&T, and Hewlett-Packard). The aberration about the defects they found was that they were not randomly distributed throughout software applications. Rather amazingly, they become clumped into localized sections, called *"defect-prone modules."* As a rule, ***about 5% of the modules in large software systems will be infected with almost 50% of reported defects.*** Once those modules are identified, the defects are usually removable. High complexity and poor coding procedures are often the cause of defect coagulations in defect-prone modules. [JONES91] If modules are kept small (e.g., 100 SLOC), it is often cheaper and faster to just rewrite the module rather than search for and remove its defects. One useful product of defect measurement is called ***"defect removal efficiency."*** This cumulative measure is defined as the ratio of defects found prior to delivery of the software system to the total number of defects found throughout development.

$$Defect\ \mathrm{Re}movalEfficiency = \frac{DefectsFound\ \mathrm{Pr}iorToDelivery}{TotalDefectsFound}$$

This indicator gives the cumulative percent of how many previously injected defects have been removed by the end of each development phase. ***Since the cost of defect removal roughly doubles with each phase, early removal must be a process improvement priority.*** With this goal, each newly delivered software product can be expected to be of significantly higher quality than its predecessor. Increases in defect removal efficiency can be accomplished through improving the code inspection and unit testing processes. *[See Chapter 14, Managing Software Development, for a discussion on developer testing.]* ***Peer inspection efficiency*** *[discussed below]* ***is***

a very sensitive indicator of delivered software quality. Table 15-3 lists the 1990 national software defect removal efficiencies measured in function points.

	MIS Projects	Systems Projects	Military Projects	U.S. Averages
Defect Potentials	4.0	6.0	7.0	5.0
Defect Removal Efficiency	75%	90%	95%	85%
Delivered Defects	1.0	0.6	0.35	0.75
First-year User Defect Reports	0.35	0.30	0.25	0.31

Copyright© SPR All Rights Reserved

Table 15-3 1990 US Software Defect Averages
(in function points)

Measuring the efficiency of defect removal requires that each defect be evaluated to determine during which phase of development the defect was inserted. To accomplish this, it is smart to have your developer provide an identifier for each line-of-code produced that tracks when (e.g., development, test, or field maintenance phases) it was entered or changed. This is especially critical with incrementally developed products to ensure the distinction between residual defects and newly injected ones. Defects made during development must be separated from those inserted during maintenance and should be related to the increment where they were injected.

REVIEWS, AUDITS, AND INSPECTIONS

Reviews, audits, and **peer inspections** provide an effective way to improve the quality of the software production process. These procedures have the ability to foster process improvement and to motivate better work. When developers know their work will be critically examined by the Government and/or their peers, they are motivated to work more carefully, either by avoiding embarrassing messy mistakes or through pride in consistently producing a quality product. [HUMPHREY90]

The RFP should request that offerors identify **formal reviews, audits,** and **peer inspections** that they plan to conduct within their software development organization and those in which the Government is invited to participate. **Reviews** and **audits** help the Government (and the contractor) in determining the contractor's technical progress against their plan relative to cost and schedule.

Peer inspections serve a different purpose by providing an effective in-house process improvement and quality control mechanism. Offerors should be required to describe the extent to which they will incorporate peer inspection techniques in their development process, that their staff has the experience to make these activities useful, and that the reviews are an integral part of the normal software development process.

Reviews and audits are a risk reduction technique used to ensure that delivered software meets the user's needs. They provide feedback and clarification to the contractor on the interpretation and implementation of requirements. They give the Government a chance to participate in user interface design, definition, and refinement and in test programs to verify software reliability and requirements compliance. Reviews and audits, usually marking a major milestone event, also provide the Government with a means to ensure software supportability. The accuracy and consistency of user and maintenance documentation must be reviewed to ensure life cycle support requirements are met. Aside from their benefits, however, the Government review and approval process is a ***major acquisition cost driver and should be used judiciously.***

Peer Inspections

Reducing the cost of rework through full-fledged **formal peer inspections** is a major evaluation criterion in the attainment of an SEI maturity **Level 3**, a **Defined Process**. In the commercial sector, where productivity and quality are critical to the survival of globally competitive software developers, inspections are widely employed. Reports from companies such as Raytheon, AT&T, IBM, and Bell-Northern Research abound with the benefits and cost savings experienced through peer inspections. NASA's **Space Shuttle** program, its Jet Propulsion Laboratory, and Goddard Space Flight Center all have quantified positive benefits through the use of peer inspections.

If you rely on testing alone to remove software defects, you will be passing about one out of every four defects on to your user. In addition, testing is almost useless in its ability to deal with front-end errors, such as those found in requirements and designs. While testing concentrates mainly on code, peer inspections can be performed on anything created by the development process that is visible and readable (such as requirements, documentation, designs, test cases, and test plans). Peer inspections are performed much nearer the point of insertion than testing, they use less resources for rework, and thus, more than pay for themselves. In fact, inspections can be applied to all phases of development to verify that key **software quality attributes** are present immediately after the point at which they should first be introduced into the product. They can also be applied to test plans and test cases to improve testing defect detection efficiency. [FAGAN86]

Peer inspections provide a formal, structured, disciplined approach to software quality control and process improvement. Because they are conducted by the software developer's peers and co-workers, they have the benefit of instilling a sense of pride in work well done. They also stimulate increased attention to detail and carefulness in performance, not necessarily present when work products are self-inspected/tested. As General Patton proclaimed, "*One of the primary purposes of discipline is to produce alertness.*" [PATTON47] Defects brought to your attention by your peers have the tendency to be well-remembered and seldom repeated. Other **peer inspection benefits** include:

- They ensure that associated team members are technically aware of theirs and each others products;
- They help to build high-performing technical teams;
- They help to use the organization's best talents;
- They provide team members with a sense of achievement and participation;
- They help the participants develop their skills as reviewers;
- They provide an orderly means to implement a standard of *software engineering excellence* throughout the development program; and,
- They pass along the lessons-learned of more experienced engineers to their junior, less experienced peers.

The basic **objectives of inspections** are to:

- Find errors at the earliest possible point in the development cycle,
- Ensure that the appropriate parties technically agree on the work,
- Verify that the work meets predefined criteria,
- Formally complete a technical task; and,
- Provide data on the product and the inspection process.

[HUMPHREY90]

Peer inspections are not to be confused with **walkthroughs,** which can be anything from casual peer reviews to management inspections. Walkthroughs usually do not employ a process that is defined, repeatable, or that collects data (whereas inspections do), and hence, they do not represent a process that can be studied and improved. [FAGAN86] Formal software peer inspections are structured events with a system of checklists and predefined roles for participants.

Cost and Quality Benefits of Inspections

Given their proven benefits, it is alarming that inspections are seldom used by software developers on large DoD software-intensive programs. A report by the **Institute for Defense Analysis (IDA)** states one reason for DoD's reluctance to jump on the peer inspection bandwagon is the increased upfront spending required — even though the downstream benefits are well documented. IDA estimates inspections require an upfront investment of up to 15% of total software development costs. ***They indicate the peer inspection investment reaps 25% to 35% increases in total productivity,*** which translates into 25% to 35% schedule savings by reducing costly rework in later phases.

Formal peer inspections are an industry-proven, verified and documented, successful method for removing defects and reducing costs. ***They can eliminate approximately 80% of all software defects, and when combined with normal testing, can reduce the number of latent defects in fielded software by a factor of 10.*** [BRYKCZYNSKI93[2]] Figures gathered from the O'Neill Software Inspections Course indicate that the implementation of a peer inspection program initially results in the detection of 50% of inserted defects. As an organization acquires inspection skills and refines its process, the detection rate increases from 80% to 90% within 18 months. [O'NEILL94]

Continuous process improvement eventually leads to fewer and fewer defects. This is a long-term commitment to quality based on a long-term evolutionary process. The question arises, *"What do you do with the defects that are in your code now until such time that your process matures and you prevent more defects than you create?"* In Chapter 14, *Managing Software Development*, you learned that ***reliance on testing alone will not remove all software defects***. Because programmers perform unit testing on their own code, ***unit testing only has a 70-75% cumulative defect removal efficiency***. [JONES91]

NOTE: Defect detection (through testing) and their removal (through rework) leads to more defects through *"bad fixes."* The Cleanroom process *[discussed below]*, **is a proven method for eliminating this source of defects.**

Studies show that **31.2%** of the defects found during system testing are inserted during the requirements analysis and design phases, **56.7%** are inserted during coding, and **12.1%** are inserted during unit testing and integration. [RAGLAND92] Figure 15-19 illustrates the number of defects that are passed on from one development phase to the next. The number of defects at the completion of unit testing, and the number of defects delivered to the field, are well documented industry averages. [JONES86]

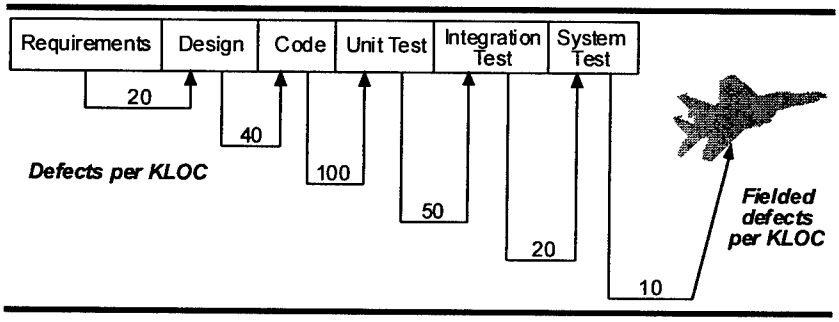


Figure 15-19 Industry Average Defect Profile [JONES86]

Figure 15-20 illustrates the number of defects passed on from one development phase to the next when inspections are used as compared to testing alone. The cumulative effect of requirements, design, and code inspections has an order of magnitude reduction in the number of latent defects in fielded products. Jones' statistics indicate that **peer inspections** produce the following results:

CHAPTER 15 Managing Process Improvement

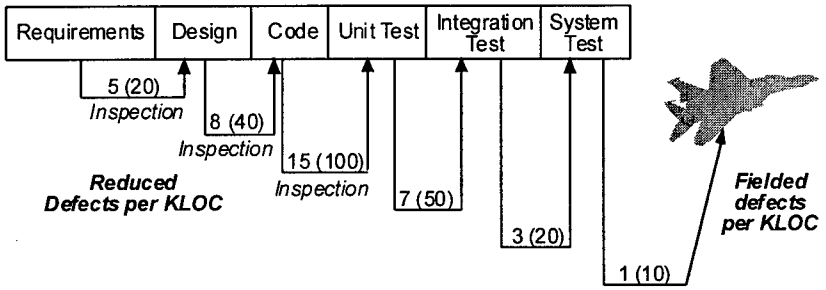


Figure 15-20 Defect Profile with Inspections [JONES86]

- Peer inspections of requirements identify **40%** of requirements errors;
- Peer inspections of designs identify **55%** of design errors and **15%** of requirements errors escaping requirements reviews;
- Peer inspections of code identify **65%** of code defects, **20%** of requirements errors, and **40%** of design errors escaping design reviews; and
- Peer inspections result in cumulative defect removals of **59%** of requirements errors, **73%** of design errors, and **65%** of code defects. [JONES91]

In addition to the gains in quality, inspections produce corresponding **gains in productivity** as the amount of rework needed to fix defects is greatly reduced. Inspections also uncover defects that are not found by testing. For example, testing alone cannot always identify special cases or unusual conditions where an algorithm produces incorrect results. [BRYKCZYNSKI93²] **Interface defects** are another example. Where one programmer's code must interface with another programmer's, the calling arguments in one module must be what the other module is expecting. If an inspection is not performed, this kind of defect can go undetected until integration or later. If left until coding, a defect of this type requires substantially more time and money to correct. *[Using Ada and compiling specifications will also catch the vast majority of these type defects.]* [RAGLAND92] Design inspections examine the logic, efficiency, and clarity of the design as represented in design documentation. Studies show that almost ***60% of software defects can be traced back to the design process.*** **Design inspections** provide detailed feedback on a relatively real-time basis. Therefore, they allow designers to experience process improvement as they progress through the design process. [AFFOURTIT92]

Studies show that companies using peer inspections tend to front load the commitment of **personnel resources** to the initial phases of development (to requirements and design). By doing this, they greatly reduce the effort required during testing and for scrap and/or rework of design and code. This results in an overall *net* reduction in development costs and schedule. Figure 15-21 illustrates the difference in resource loading against the time schedule between software development with and without inspections.

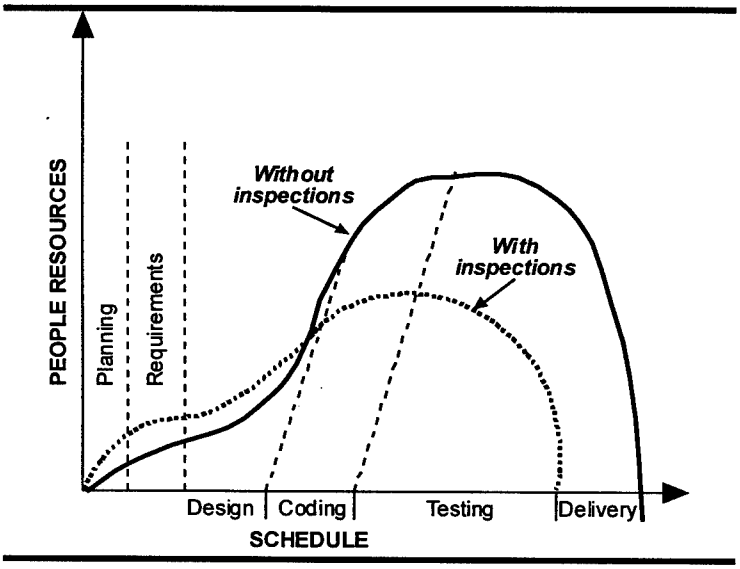


Figure 15-21 Personnel Resource Expenditures With and Without Inspections

Formal Peer Inspection Process

Inspection team members must prepare in advance, and have their concerns and questions identified before the inspection starts. The purpose of inspections is to identify problems, ***not*** to resolve them. Follow-up and defect removal are the responsibility of the inspected product's creator. These strict procedures, combined with checklists and standards, ensure that inspections are conducted within a minimum time frame. Generic checklists and standards are developed for each inspection type and tailored to specific program needs. These checklists cover inspection planning, preparation, conduct, exit and reporting criteria. [HUMPHREY90] **Exit criteria** are the standards against which inspections measure the completion of a work product at the end of a development operation. They verify the presence or

absence of the **quality attributes** that satisfy the requirement(s). *[A deviation from an exit criterion is a defect.]* [FAGAN86] Inspections are conducted by technical personnel for technical personnel. Managers do not attend, but are provided with the findings and the dates when identified problems will be/are resolved. Inspection data must be entered into a database and used to monitor inspection effectiveness and track product quality. [HUMPHREY90]

The inspection consists of a defined series of steps, such as overview, preparation, inspection, rework, and follow-up, as illustrated by the **Fagan Inspection Process** on Table 15-4. The inspected work product is relatively small, sometimes only 4 to 5 pages of code. The inspection meeting is usually attended by a small number (4 to 5) co-workers and lasts less than two hours. Because inspections are rigorous, exacting technical work, defect detection efficiency in meetings over two hours tends to drop considerably. [FREEDMAN90] A task is assigned to each inspector, which Fagan defines as:

	ACTIVITY	OBJECTIVE(S)
1	Planning	• Materials meet entry criteria • Assign inspector roles • Schedule next 4 activities • Meeting places (activities 2, 4, & 5)
2	Overview	• Educate inspection team - Background/context/rationale
3	Preparation	• Each inspector learns material - Sufficient to maintain the product • Prepare to fulfill role
4	Inspection	• FIND DEFECTS - No solutions or improvements
5	Process Improvement	• Continuous improvement of development process (systematic defect removal) • Improve inspection defect detection
6	Rework	• Fix all defects and resolve investigated items
7	Follow-Up	• Verify all defect fixes and proper resolution of all investigated item
	Copyright © 1995 Michael E. Fagan	

Table 15-4 Fagan Inspection Process Activities [FAGAN95]

- **Moderator**. The moderator manages the inspection team while playing an active role as an inspector. He must be competent in the type of work being inspected and have the ability to be tactful, diplomatic, forceful, and to work well with others.
- **Author**. The author is the creator of the work product being inspected. He has a vested interest in ensuring that the inspection finds all possible defects and the product meets exit criteria without ambiguity.
- **Reader**. The reader is usually someone who depends on the inspected product to perform their work. During the inspection, readers paraphrase each statement in their own words, expressing the meaning of each statement with a level of understanding sufficient to demonstrate that they can carry out the next stage of development or fully use the work product.
- **Tester**. The tester considers how he would test the product — what paths, data, states, conditions, what expected results, etc., — and express test cases as questions. [FAGAN95]

The work product author is responsible for post-inspection defect removal and corrections. Suggested improvements are provided by other team members during the inspection meeting. **Defect causal analysis** is usually performed by a **process action team** after an inspection to identify process improvements that will prevent future occurrences of the same class of defect. [BRYKCZYNSKI93[2]] *[Consult a reference such as Humphrey's Managing the Software Process for more information on how to implement an inspection program.]* [HUMPHREY90]

Peer Inspection Case Study

A case study was performed on an engineering organization that manufactures products for other engineers. Their software is written by engineers (mechanical and electrical) who see testing as an integral part of the engineering process. Their products are highly specialized and semi-experimental. The users operate their products on a daily basis and call in problem reports. The study separated defects (did not meet requirements) from enhancements (new requirements). The study covered three groups of 30-35 engineers each. Each group was trained using a different inspection methodology; however, the amount of testing remained constant throughout the study. All the groups discovered numerous defects prior to testing, which meant few defects were found during testing. The amount of user-discovered defects remained the same as before implementing an inspection

process for two of the groups. While, the third group experienced an 80% reduction in user-discovered defects.

The study concluded that you can tell at what SEI CMM[SM] Level an organization is by looking at its inspection checklists. Inspection checklists for Level 1 organizations contain things like format, comment readability, initialization of variables. The two groups which did not show improvement in user-discovered defects use a Level 1 inspection checklist in which they looked for things an automated test tool could have checked during unit testing. Inspection checklists used by Level 5 organizations focus on timing and interface problems usually not discovered until system integration. The third group used a Level 5 inspection checklist, which focused on understanding how a given module fit into the overall system, requirements traceability, and user profiles/scenarios. This required the third group to review artifacts (similar to OT&E) not normally reviewed during software inspections. The study arrived at the following conclusions:

- An inspection process will reduce the amount of defects discovered during testing and allow defects to be corrected sooner and cheaper.
- An inspection process' ability to reduce the number of user-discovered defects depends heavily on its goals, format standardization, or early detection of errors an automated test tool would have trouble finding (e.g., determining if a module satisfies user operational requirements, interfaces with other models, or is devoid of timing problems).
- The number of tests conducted should not be lower as a result of implementing an inspection process. The time it takes to conduct the tests could be less than without inspection, if testing finds fewer defects, which in turn, result in writing few problem reports.

[SONNEMANN94]

Peer Inspection Buy-In

Current DoD acquisition guidance includes peer inspections as a recommended approach to software quality. However, less effective methods such as walkthroughs and informal reviews may also be used. This latitude in acceptable approaches makes it relatively easy for contractors to avoid peer inspections altogether. To benefit from this proven method of defect detection, removal, and prevention, ***you must make sure your developer has a formal, institutionalized, peer inspection process in place.*** You must also make sure you have your contractor's full cooperation and *buy-in* on the value and

use of formal peer inspections. Contractor management and technical personnel often resist peer inspections because the process requires that a software developer's work be exposed to very intense group scrutiny. However, industry studies confirm that ***formal peer inspections are efficient ways to remove software defects, especially for ultra-large software developments.*** [RUSSELL91]

Peer Inspection Training

Do not forget that the application of effective software inspections requires **training**. There are a number of industry training courses available. *[O'Neill Software Inspections Course and Loral (formerly IBM-Houston) on contract to the Air Force are two sources. Contact the STSC for more information. (See Volume 2, Appendix A for information on how to contact these sources.)]* Companies that provide inspection training rely on repeat business, and often train their customers' entire software staff, including management. [BRYKCZYNSKI93[2]]

ATTENTION! If you are managing an on-going program for which inspections would be beneficial, consider an investment in this training for your team.

Independent Verification & Validation (IV&V)

IV&V is a process for evaluating software and associated products for compliance with requirements and specifications by a third party. In February 1992, the Air Force Inspector General published a report on the effectiveness of software IV&V within the Air Force. The results of this report showed that IV&V is an effective tool — but poorly utilized. It also found that, in programs where it was applied, it was performed inconsistently. [TIG92] Determining whether to implement IV&V in your program must depend on the degree of risk you identify. For guidance on IV&V, consult NIST Special Publication 500-165, *Software Verification and Validation*, and ANSI/IEEE Standard 1012-1986, *IEEE Standard for Software Verification and Validation.*

CLEANROOM ENGINEERING

Software engineers have always sought to take advantage of evolutionary advances in technology to achieve revolutionary improvements in quality and productivity. The best technological approach of today quickly becomes outmoded in the wake of an innovative, superior process. In the engineering of software, several DoD organizations are demonstrating an approach to software engineering that is revolutionizing traditional software development practices. This approach is called ***"Cleanroom engineering,"*** which was adopted from the hardware industry to emphasize the concept of an engineering environment devoid of contaminating factors that adversely affect product quality. Like a contaminant-free hardware environment, the Cleanroom process emphasizes ***preventing defects***, rather than removing them after the fact. [CLEAN90[2]]

The Cleanroom approach differs from traditional software engineering approaches in that it is a theory-based, team-oriented process for development and verification of ultra-high-reliability software systems by improving productivity through statistical quality control. Cleanroom delivers software with a known and certified **mean-time-to-failure (MTTF)**. It combines practical new methods of specification, design, correctness verification, and statistical testing for certifying software quality using a process based on incremental development, as summarized on Figure 15-22. Note that the formal

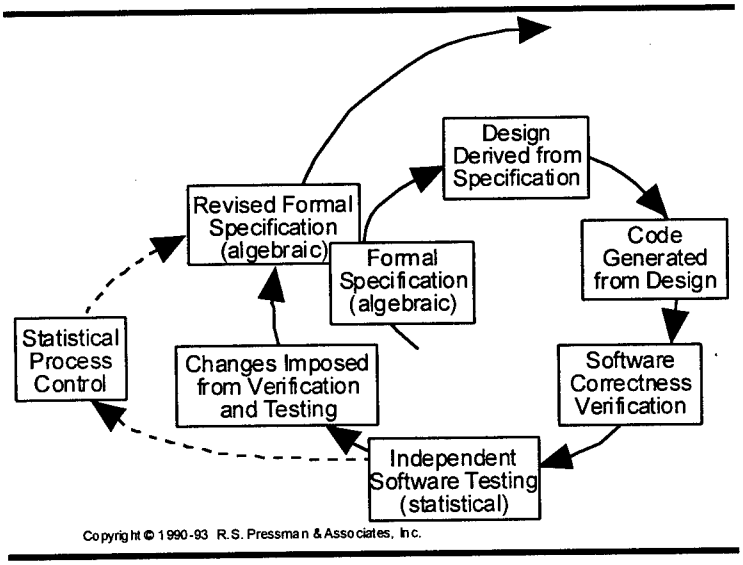


Figure 15-22 Cleanroom Process Model [PRESSMAN93]

specification is algebraic, and correctness verification is used in place of unit testing and debugging where all defects are accounted for from first execution.

Cleanroom development teams produce software approaching zero-defects through the use of a rigorous stepwise refinement and verification process for specification and design using object-based *Box Structure* technology. Box Structures permit precise definition of required user functions and system object architectures which scale up to maintain intellectual control of large software developments. ***With Cleanroom, correctness is built-in, not tested in!***

Cleanroom achieves statistical quality control over software development by strictly separating the design process from the testing process. Unlike the traditional waterfall, Cleanroom management is based on development and independent certification testing of a pipeline of user-function increments that accumulate into the final product, as illustrated in Figure 15-23. System integration is top-down with system functionality growing through the addition of successive increments. When the final increment is complete, the system is complete. At each stage, the harmonious operation of future increments at the next level of refinement is predefined by increments already in execution; therefore, interface and design defects are rare.

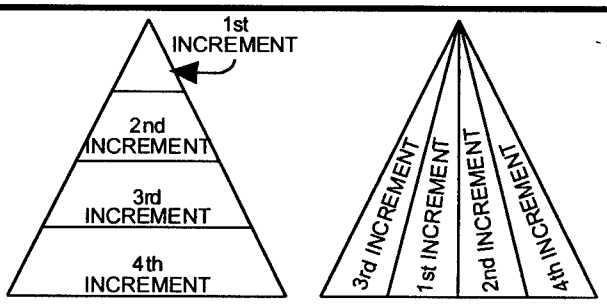


Figure 15-23 Cleanroom Pipeline Construction

Cleanroom applies a top-down, stepwise refinement of the total design, with correctness verification required at each refinement step. Cleanroom programmers deal with a complete set of eventualities in anticipation of the statistical testing of their code (not found in traditional methods) that are beyond their choosing or control. This

leads to a level of design precision and completeness that is thought through at each refinement step. Mainline processing and exception handling *[discussed in Chapter 5, Ada: The Enabling Technology]* are addressed upfront because statistical testing requires that the software perform with all possible inputs, both correct and incorrect, at each increment. Incremental development enables early and continual user feedback and assessment of product quality, and facilitates improvements as development progresses. The incremental approach permits controlled, stepwise integration and quality certification of components — avoiding the risky eleventh-hour integration. The result is a more coherent, systematic design with complete required behavior built-in at each refinement, not added on after the fact.

Correctness Verification

Cleanroom-developed software is subject to rigorous verification by development teams prior to release to certification test teams. A practical and powerful process, correctness verification permits development teams to completely verify the software with respect to specifications. A ***"Correctness Theorem"*** defines conditions to be met for achieving zero-defect software. These conditions are confirmed in special team reviews — through group reasoning and analysis that result in "*mental-proofs-of-correctness*." Even though programs of any size contain virtually an infinite number of paths, the theorem reduces verification to a finite number of steps and ensures that all software logic is completely validated in all possible circumstances of use.

Unit testing and **debugging** by programmers is not part of Cleanroom because they compromise the correctness of the original design and introduce complex software defects from the *"tunnel vision"* inherent in the debugging process. In unit testing, programmers often focus on getting mainline code working first, testing other parts (such as exception handling) only if time permits. Treating any part of program logic as an afterthought can lead to incomplete solutions delivered with little or no testing.

NOTE: See Chapter 14, *Managing software Development*, for a discussion on the inadequacies of unit, integration, and system testing.

The effectiveness of correctness verification drives down the cost of quality by eliminating unnecessary testing, debugging, and rework. The verification process is far more powerful than unit testing in eliminating defects and is a major factor in the dramatic quality improvements experienced by Cleanroom teams. Correctness verification results in software of sufficient quality to enter system testing directly with no prior execution by the development team.

Cleanroom certification teams mathematically certify software reliability—they do not test it in. Following correctness verification, software increments are placed under engineering change control and undergo first execution. Statistical usage testing is performed to produce scientifically valid measures of software quality and reliability by testing software the way it is intended to be used. Test cases are built based on **usage probability distributions** that model anticipated use in all possible circumstances, including unusual and stressed situations. Objective statistical measures of software reliability, such as MTTF, are computed based on test results. Because Cleanroom statistical testing is based on random sampling driven from input probability distributions, independent of human selection, it is uniquely effective at finding first those more serious, high-frequency defects with high failure rates first. It is, thus, far more effective at improving software reliability in less time than traditional testing methods.

The Cleanroom test team is responsible for validating software quality with respect to its specification. If the software's quality is not acceptable, it is removed from testing and returned to the development team for rework and reverification. [DYER92] Figure 15-24 illustrates the Cleanroom certification process.

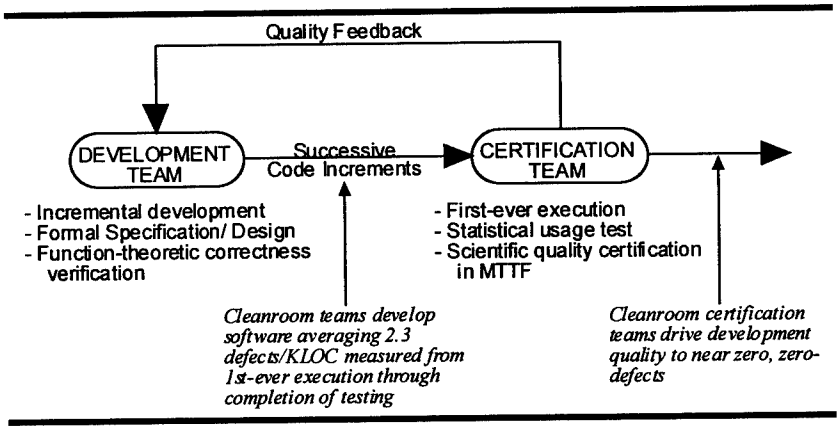


Figure 15-24 Cleanroom Certification Process [LINGER94]

Cleanroom Engineering Results

The Cleanroom process can be applied to the development of new systems, maintenance and evolution of existing systems, and salvage and re-engineering of problem systems. ***It is language, environment, and subject-matter independent and can be used to develop and evolve a variety of systems***, including real-time, embedded, host, distributed, workstation, client-server, and microcode systems. Cleanroom is compatible with object-oriented and prototyping techniques, and promotes reuse through precise definition of component functional semantics and certification of component reliability. Cleanroom has been adopted by over 30 development organizations who have never experienced a Cleanroom program failure. Examples of DoD Cleanroom programs include:

- **Air Force STARS Demonstration Project.** The USAF Space and Warning Support Center at Peterson AFB, Colorado is integrating the SET, Inc.'s Cleanroom engineering process and TRW's **Ada Process Model** to use the **Reusable Integrated Command Center (RICC)** architecture tools. This combination of methodology and tools are being used to develop a Space Command and Control application. The **Rational Apex™** product set *[discussed in Chapter 10, Software Tools]* is also being used to support this integration process.
- **Army STARS Demonstration Project.** The Army Life Cycle Software Engineering Center at Picatinny Arsenal is using the Cleanroom process on two programs, the **Conduct of Fire Trainer (COFT)** and the **Mortar Ballistics Computer (MBC)**. On these programs, the development team achieved productivity gains over their baseline by a factor of 4. Quality (in terms of failures observed from first execution) increased substantially to less than one failure per KLOC. An accounting of the costs to perform the program, and the process improvement costs incurred to transfer the technologies to the team, was maintained so process improvement return on investment (ROI) could be calculated. The ROI was in excess of a factor of 10.
- **Naval Coastal Systems Station**. The AN/KSQ-1 development team is applying the Cleanroom process to develop an embedded software system that provides a bridge between workstations and the **Position Location Reporting System**. [SET93]

Since late 1993, experience with one million lines of Cleanroom-developed software from a variety of programs has shown extraordinary quality compared to traditional results. *[The million lines-of-code exhibited a weighted average of 2.3 defects per thousand lines-of-code (KLOC), measured from first-ever execution to the completion of all testing]* [HAUSLER94] As shown in Table 15-5, this defect rate compares to 50-60 defects/KLOC for traditional, software-as-art practices and 20-40 defects/KLOC for contemporary software engineering practices. [DYER90]

SOFTWARE DEVELOPMENT PRACTICES	DEVELOPMENT DEFECTS/KLOC	OPERATIONAL FAILURES/KLOC	PRODUCTIVITY LOC/STAFF MONTH
Traditional Software-as-art	50-60	15-18	Unknown
Software Engineering	20-40	2-4	75-475
Cleanroom Engineering	0-5	<1	>750

Table 15-5 Cleanroom Performance Measures
(KLOC = 1,000 lines-of-code)

Highlights of Cleanroom programs with defect rates measured from first-ever execution of the software to completion of all testing are listed here. These developments have been produced by first-time Cleanroom teams composed of journeyman programmers.

- **IBM COBOL Structuring Facility.** A six-person team developed a 85-KLOC product which automatically structures unstructured COBOL programs and experienced only 3.4 defects/KLOC in all testing. The product experienced 7 minor defects in the first three years of field use (all simple fixes) demonstrating the dramatic reduction in maintenance costs associated with Cleanroom products.
- **Ericsson AB Telecommunications Operating System.** A 73-person, 18-month, on-schedule development of a 350-KLOC operating system for switching computers resulted in 1.0 defect/KLOC in all testing. Productivity improved 70% in development and 100% in testing. They announced that on the development of a new operating system to support real-time applications they increased productivity by nearly 100% and decreased failure rates by more than a factor of 5. These improvements meet an organizational baseline, they claim, meets or exceeds their competitors in the telecommunications industry. They are

accelerating their adoption of Cleanroom so they can more fully engineer future software solutions.

- **IBM AOEXPERT/MVS.** A 50-person team developed this 107-KLOC complex, systems management product and experienced 2.6 defects/KLOC during testing. No defects were reported from beta sites and early users.
- **NASA Satellite Control Projects.** NASA's **Goddard Space Flight Center Software Engineering Laboratory** has completed its second and third Cleanroom programs, a 20-KLOC attitude determination subsystem and a 150-KLOC flight dynamics system that experienced a combined defect rate of 4.2 defects/KLOC in all testing.
- **IBM Tape Drive Microcode Project.** A five-person team developed an 86-KLOC embedded system for processing real-time data streams in a multiple-processor bus architecture. The program experienced 1.2 defects/KLOC in all testing. A total of 490 statistical tests were executed against the final version of the product with no defects found.

Cleanroom for New-Start Programs

While Cleanroom incorporates technologies with strong theoretical foundations, it also has strong techniques for management, development, and testing which ***have been simplified for practical application and are effective in everyday use by journeyman programming teams.*** Cleanroom provides a coherent management and technical framework for intellectually-controlled, on-schedule software production. Incremental development permits early quality assessment and user feedback on system function and avoids the risk associated with 11th-hour component integration often experienced in traditional developments. The high quality of Cleanroom-developed software results in substantial reductions in maintenance and support costs.

The Cleanroom process can be integrated into DoD acquisition practices in terms of required program processes and deliverables. For example, statistical reliability certification can be specified, together with an incremental development process and deliverables associated with rigorous software specification, design, correctness verification, and certification.

Cleanroom for On-Going Programs

Modifications and extensions to existing software and non-Cleanroom software can be developed with Cleanroom specification, design, verification, and certification technology. In addition, problem-prone modules in existing systems can be re-engineered to Cleanroom quality through use of design abstraction and correctness verification techniques.

Cleanroom for Troubled Programs

Cleanroom specification, design, verification, and certification technology provides a framework for objective status assessments of troubled software programs. It can be used to salvage and re-engineer partially-completed components into intellectual controlled, coherent structures. Incremental development provides a framework for management control in salvaging software assets and for the introduction of disciplined development processes.

Cleanroom Training

The Cleanroom process can be introduced through short, skills-based courses combined with consultation support by experienced Cleanroom practitioners. *[For more information, contact the STARS program office or STARS reports, such as, "Cleanroom Reliability Manager: A Case Study using Cleanroom with Box Structures ADL." [STARS90] Overviews of the Cleanroom process and its introduction can be found in Hausler and Linger.]* [HAUSLER94] [LINGER94]

Cleanroom Information

If considering Cleanroom, is it strongly recommended you obtain the STSC's Cleanroom Pamphlet, April 1995. *[See Volume 2, Appendix A for information on how to contact the STSC.]* It contains the following:

- Linger, Richard C., "Cleanroom Software Engineering: Management Overview," Software Engineering Institute
- Hausler, P.A., Richard C. Linger, and C.J. Trammell, "Adopting Cleanroom Software Engineering with a Phased-Approach," *IBM Systems Journal*, Vol.33, No. 1, 1994
- Linger, Richard C., "Cleanroom Process Model," *IEEE Software*, March 1994

- Sherer, Wayne S., Paul G. Arnold, and Ara Kouchakdjian, "Experience Using Cleanroom Software Engineering in the US Army," *Proceedings from STC '94* (updated)
- Henderson, Johnnie, "Why Isn't Cleanroom the Universal Software Development Methodology?"
- Bibliography of Cleanroom articles/books
- Listing of organizations that assist with Cleanroom adoption

IMPROVING PRODUCTIVITY

> *It is more than probable that the average man could, with no injury to his health, increase his efficiency fifty percent.*
> — Sir Walter Scott

There is no denying that software technology has been the greatest instrument for improving man's efficiency since the Industrial Revolution. When properly used, it provides remarkable competitive advantages. The paradox is, while software significantly increases the efficiency of its users, the way software is produced is usually quite inefficient. Software development remains ***extremely human-intensive***, because the production process is imbued with human limitations (i.e., software is, and will be for the foreseeable future, largely hand written). The bottom line, the goal, for everything discussed in this chapter on process improvement is to ***increase productivity.***

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on measuring productivity and productivity cost drivers.

The reason we want to remove and prevent defects early is to eliminate the time wasted in scrap and rework of defective code. We want to build statistically controlled Cleanroom software to minimize the need for time-consuming unit testing. We want to produce modifiable, understandable, well-documented software to eliminate the costly wheel-spinning that occurs during fielding. All these process improvements are aimed at increasing productivity, which shortens the time to field, saving development dollars. Process improvements (such as automated tools, reusable software components, iterative refinement of prototypes through continuous user input and buy-off) are also key factors for high productivity. As technology advances, the way these processes are employed will also increasingly improve. [FISHER91]

Boehm tells us that to stabilize the process, reduce team communication overhead and ripple effects, reduce risk, minimize requirements changes, specify interfaces more precisely, and capitalize on the software engineering methods promoted by the **Ada language**, the following set of guidelines should be followed.

- Use a set of risk management plans to drive the process, using incremental development to stabilize the process.
- Try to get the uncertainties out of the requirements process, partly by prototyping.
- Do not force your developer into a PDR in 90 days.
- Do not force your developer into having to build 52 documents in 90 days (i.e., the less documentation required, the higher the productivity).
- Do not allow your developer to overload the team with a huge number of people.
- Do not use document milestones to drive program organization.
- Do not force developers into having separate requirements and design teams; they will eventually have to integrate them.
- Do not force every development into the same sequence.
- Do not wait around passively for someone to provide definitive requirements.
- ***Software people have to get involved in the systems engineering process.*** *"That,"* Boehm claims, *"is the biggest leverage item of them all."* [BOEHM89]

Other basic ways to improve productivity include:

- Using Ada *[discussed in Chapter 5, Ada: The Enabling Technology]*,
- Software development maturity *[discussed in Chapter 7, Software Development Maturity]*,
- Reuse, *[discussed in Chapter 9, Reuse]*,
- Design simplicity *[discussed in Chapter 14, Managing Software Development, and in Chapter 8, Measurement and Metrics, under "Size" and "Complexity,"]*
- User involvement *[discussed in Chapter 14, Managing Software Development]*
- Prototyping *[discussed in Chapter 14, Managing Software Development]*,
- Automated tools *[discussed in Chapter 10, Software Tools]*, and
- Training *[discussed below]*.

Ada Use

The methods for increasing productivity must be selectively applied based on your development needs. ***One basic way for improving productivity is by using Ada.*** Figure 15-25, based on data collected by Reifer, illustrates the difference in time spent by programmers on Ada programs versus non-Ada programs. With the substantial portion of development time spent on design, the effort is focused on building-in quality, thus avoiding the productivity nullifiers of defect removal, scrap, and rework in later stages.

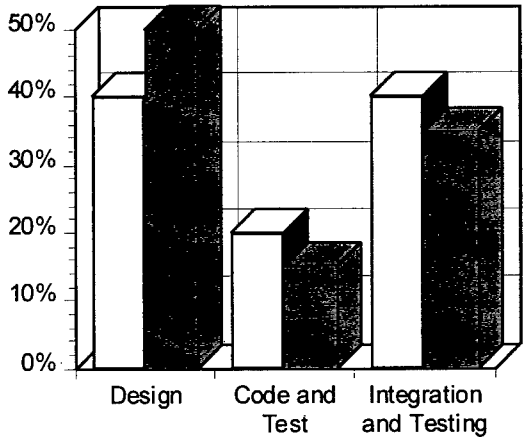


Figure 15-25 How Programmers Spend Their Time [HORTON89]

NOTE: As mentioned in Chapter 5, *Ada: The Enabling Technology*, there is an anomaly concerning Ada and productivity. With other languages, as software size increases, productivity decreases. The experts concur that the surest way to increase downstream software productivity is to decrease the number of SLOC to be delivered. With Ada, however, as software size increases so does productivity. This relates in large measure to internal code reuse (i.e., within the Ada program) of about 30%.

Reuse

As discussed above, **reuse** is an effective technique for increasing productivity by reducing the number of source code instructions to be produced. One way to enhance the possibility of reuse is through the concept of **object-oriented development (OOD)** *[discussed in Chapter 4, Engineering Software-Intensive Systems]*. By basing the system architecture on the characteristics of data rather than functions, the software tends to be more robust, easier to maintain, and easier to reuse.

Design Simplicity

Software size (number of deliverable source code statements) and complexity have enormous impact on productivity. Therefore, design simplicity is a productive strategy. When specifications are too abstract, users have trouble understanding them because they cannot imagine what the new system will be like. Through feelings of uncertainty, and not being able to visualize what they will be getting, users often derive additional requirements. Because they do not know enough about the user's environment, the developer will incorporate all the new requirements into the system. Often these requirements become superfluous, unnecessarily inflating development costs. This type of over-specification is sometimes called *"**goldplating**,"* a great hindrance to software development productivity. Thus, it is important that specifications be simple and clear enough that users do not feel obliged to *goldplate* in an attempt to gain functionality they might already have — but cannot divine from the specification. [GLASS92] *[Prototyping also guards against goldplating.]*

NOTE: Keep your Ada software modules small with clearly defined inputs and outputs — as if you were designing dependable and discrete components for inclusion on a printed circuit card. [MOSEMANN94]

User Involvement

Software designers usually lack the detailed knowledge required to understand the subtle aspects of the user's problem environment to design the optimal system. Since user satisfaction is the main criterion for software quality and program success, ***user involvement is extremely necessary***. There is a direct correlation between user

involvement in design and user satisfaction. IBM conducted a study on user participation in design using a joint design methodology (as compared to traditional design techniques). Productivity was measured in function points. Their user-participative design method produced productivity improvements of 50%. [GILL90]

Prototyping

One way to avoid unnecessary complexity in development is through prototyping (and demonstrations). It is the best method for solving the problems associated with abstract software specifications. The user can see and experiment with software that really works. They can understand how the software will fulfill their needs, and then make useful and practical suggestions. Working with prototypes is the most effective way to train users and to get them productively involved in the development process. *[Prototyping is discussed in detail in Chapter 14, Managing Software Development.]*

Automated Tools

Improving productivity requires making the programmer's job easier. CASE tools provide easy ways to do this, but their selection must depend on detailed knowledge of programmer activities. Tool selection should be based on a needs-driven process with the help of the people who have to use them. If tools do not fit into the current process, the training required to learn new processes, or bend the old ones to fit the tool, can be costly. *[Tools are discussed in detail in Chapter 10, Software Tools.]*

Maintenance of test environments is another important productivity enhancer gained by tools. Programmers spend a great deal of time building and rebuilding test environments. A well-organized test environment saves tremendous time during unit testing, systems testing, systems reviews, and quality control. [FISHER91] CASE tools are often used to produce graphic representations of systems specifications for the user's review. Although CASE tools are great productivity enhancers, if improperly applied, they can have negative impacts. The difficulty with many CASE products is that they produce abstract graphic representations that can be as difficult for users to understand as traditional specifications. CASE diagrams often help designers communicate better among themselves, but not with the average user. Excessive automatic diagram generation can have the effect of overwhelming the user with stacks of specifications. Users then become less able to visualize how the system will work,

and are less effective participants in its development. Your developer must first decide on a specific development methodology, and then look for those tools that are necessary to accomplish the specific needs that evolve. ***Excessive complexity in the development process is inordinately counterproductive.***

Software Productivity Consortium

Founded in 1985, the **Software Productivity Consortium (SPC)** represents one of the nation's most innovative and unique approaches to improving the processes and methods needed to develop and deploy major software-intensive systems. Consortium technologies in systems and software engineering focuses on continuous process improvement, risk mitigation, measurement, design, reuse, and technology insertion. It provides a common forum for software and systems engineers in the defense and civilian government agencies, the aerospace, defense, and systems integration industries, and the academic community. Its products and services help members achieve the higher levels of software maturity defined by the SEI's CMM[SM] *[discussed in Chapter 7, Software Development Maturity].*

Through an on-going series of technology and executive forums, workshops, training courses, round tables, user group meetings, and other events focused on key issues of concern to the systems and software engineering community, the consortium serves as a fulcrum by which to leverage our efforts to advance the productivity, quality, and capability of the systems and software engineering process. The consortium technical program is tightly focused on advancing the fundamental processes and methods required to build software-intensive systems competitively. This focus is timely, as industry experience increasingly demonstrates that organizations without such software and systems engineering processes and methods can expect only marginal productivity gains through the use of automation technologies alone. Consortium products include:

- The **ADARTS[SM]** design method for real-time systems, currently the standard design method for avionics software on the **F-22** and used by many other programs.
- The **Evolutionary Spiral Process (ESP)**, a life cycle decision management and risk mitigation process in use on the Air Force **Cheyenne Mountain Granite Sentry C2 upgrade**, the Army's All Source Analysis System, and other programs.

- The **Consortium Requirements Engineering** method for analysis and specification of real-time system requirements, used on the **C-130J** upgrade by Lockheed.
- The Reuse-driven Software Process methodology, used on the STARS/Boeing/Navy T-34C flight trainer program.
- Several process improvement, measurement, and technology insertion processes and methods.

NOTE: Contact the consortium *[see Volume 2, Appendices A and B]* **to receive their Catalog of Products and Services.**

TRAINING

> *In any epoch, the difference between a rabble and an effective professional Army is training. No task is more important than training as we face this decade.*
> —General Edward C. Meyer

In this decade, ***the difference between rabble software developers and world class software engineers is training!*** This axiom was disclosed in a report from the Standard Systems Center, Gunter AFB, Alabama and the Communications Systems Center, Tinker, AFB, Oklahoma on lessons-learned from their Ada and software engineering programs, requested by SAF/AQK in March 1993. According to **Brigadier General George P. Lampe**,

> *The overall message in both inputs is the need for a mature software process guided by qualified software engineers. Sound software engineering principles, such as requirements management, project management, and configuration management are the basis for successful projects. Most of the lessons-learned relate directly to these principles. It is essential for educated software engineers to lead projects to increase the probability of success. Without* ***trained software professionals and a mature process****, projects risk failure.* [LAMPE93]

General Lampe went further to say that they found the two primary nontechnical concerns with Ada were training and reuse. Because **Ada programmers** require more training on some of the language's intricacies, program mangers must be prepared to allow for the learning curve. This applies to both sides of the joint government/

industry development team. To be equipped with the highest quality software engineering and management professionals, training is one of the most critical issues you face.

It is incumbent upon all managers (government and industry) to identify the software engineering requirements in their organization and to actively pursue software engineering education and training for personnel filling these positions. Establishing, educating, and maintaining a corps of software engineers is critical to the successful implementation of software policy objectives. Thus, you must choose software engineers who have completed the course work necessary for competency within your program domain. For contractor personnel, it is important that offerors propose a comprehensive training program for their development team once awarded the contract. It is recommended that your Air Force liaison staff assigned to the contractor's facility also be included in these training programs. Better yet, seek contractors whose teams have already been well-trained.

NOTE: See Volume 2, Appendices A and B for sources of training and education.

Be aware, many offerors will claim their software professionals are highly skilled — but few development teams are adequately trained to use the language and tools required of a new development effort. Additional training is necessary to:

- Understand the protocols, system services, and software processes needed to manage the multi-vendor team of a major software-intensive software acquisition;
- Understand the unique subtleties of the application they are to develop and implement; and
- Plan, measure, track, control, and improve their own work in a standardized, disciplined fashion.

Without plans, funding, and scheduling for required training, learning will be gained through *trial-and-error*, which not only wastes time and money, but often involves substantial ***error***. [HUMPHREY90] General George S. Patton, Jr. summed up why team training is so important when he said,

A pint of sweat will save a gallon of blood. [PATTON47]

CHAPTER 15 Managing Process Improvement

The pint of resources committed to team training will save a gallon of blood in development time and money spent detecting and removing errors and defects from virtually every software artifact produced.

TRAINING IS NOT A FRILL; IT IS A NECESSITY! Assure adequate time and funding for superior training. Also, consider employment of "*coaches*" (i.e., subject matter experts) to enhance the learning experience and to consult as the development progresses.

If an integrated government/industry development team is planned, training for the team must be planned for, in addition to the training of government users/maintainers. You must require that offerors submit a **Training Plan** as part of their proposals, or as a deliverable after contract award. The cost of this requirement must also be addressed in the offeror's cost proposal. When evaluating proposed training programs, be aware that training is often wasted because it is given on some arbitrary schedule without regard to when the people needing the training are in a position and at the stage during development to actually use and benefit from it. IBM developed a **Just-in-Time training program** that addresses this issue. Figure 15-26 (below) illustrates one phase of their training program, the team launch (execution phase). Other Just-in-Time training programs are implemented for subsequent software development phases.

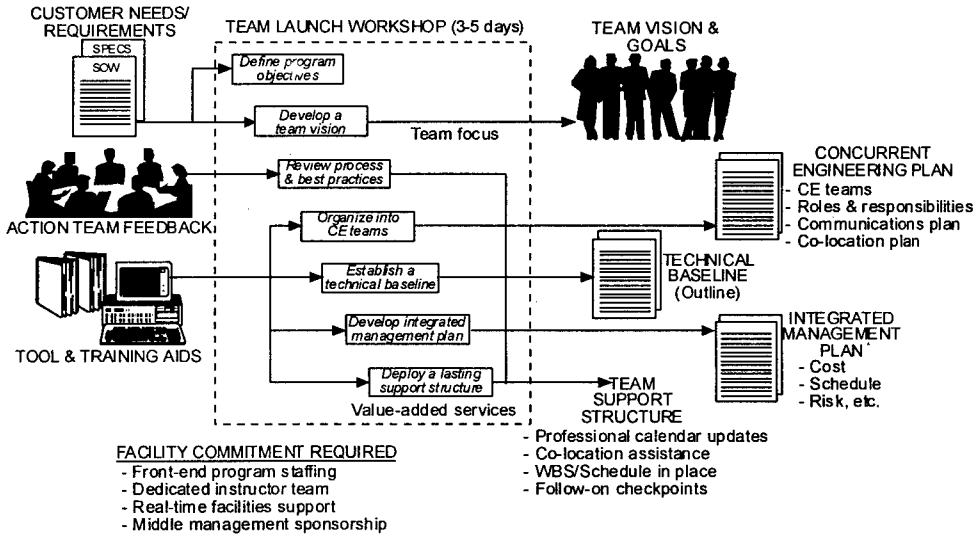


Figure 15-26 IBM's Just-in-Time Training Approach

Lessons-Learned from SSC and CSC

NOTE: See Chapter 5, *Ada: The Enabling Technology*, for a description of the programs upon which these lessons-learned are based.

- *Blanket training* should be used as it gives all team members needed insight into the overall development process and to better understand the impact of their area of specialization. *Formal* and *on-the-job training* should concentrate on each individual's tasking. This process of creating specialists produces a core of expertise.
- Training of other than program office personnel (i.e., users and maintainers) on the development process needs to be incorporated into the program schedule. DoD counterparts must be made aware that the impact of this training on the development schedule is not trivial.
- The use of the integrated team and training schedules tends to result in classes composed of individuals with diverse backgrounds and software expertise. By soliciting feedback from the students, course instructors and managers can ensure objectives are met for all students.

NOTE: See Volume 2, Appendix O, Chapter 15 Addendum B, "*Training — Your Competitive Edge in the 90s*," by Eileen Quann.

CONFIGURATION MANAGEMENT

> *I have known commanders who considered that once their plan was made and orders issued, they need take no further part in the proceedings, except to influence the battle by means of their reserves. Never was there a greater mistake. The modern battle can very quickly go off the rails. To succeed, a C-in-C must ensure from the beginning a very firm grip on his military machine; only in this way will his force maintain balance and cohesion and thus develop its full fighting potential.* — Field Marshall Montgomery [MONTGOMERY58]

Indeed, the modern software battle can very quickly go off the rails! To succeed in software management you must also ensure from the very beginning that you have a very firm grip on your software

machine. **Configuration management (CM)** provides that essential element of control at the heart of the development process. It maintains and controls the product baseline, relates the life cycle to the development process, and maintains balance and cohesion among the process, its products, and its producers. Figure 15-27 illustrates how CM is a main ingredient in the **Space Shuttle** software development process.

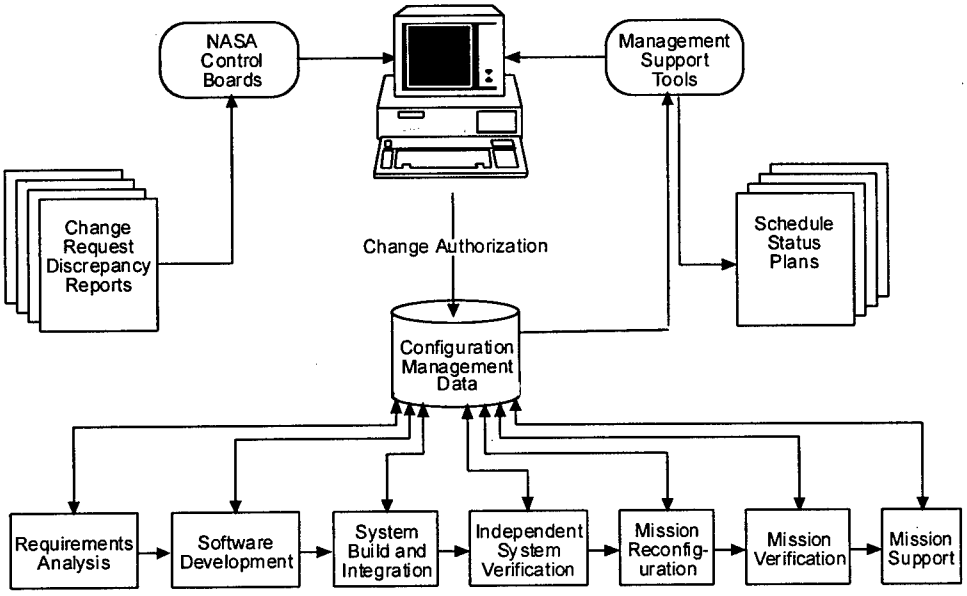


Figure 15-27 Space Shuttle Software Development Process

The configuration management task is, in theory, quite simple. In software, it embodies the practice of disciplined change control, technical and administrative guidance, surveillance, and a historical database of program progress. It defines the configuration of the software system, at discrete points in time, by systematically controlling changes and by maintaining configuration integrity and traceability throughout the system life cycle.

Configuration management facilitates communication among development team members on the status of documentation and software engineering efforts. It provides management (both Government and contractor) an identification, control, and accounting system for changes to functional, allocated, and product baselines. ***Configuration management is the management of change***. Your

RFP should require that offerors provide a **Configuration Management Plan** that addresses change control throughout the development process. This plan discusses the offeror's configuration management organization, the tools to be used, configuration management personnel experience, and a description of their configuration management training program. This plan should be evaluated against the program needs identified by the program office during acquisition planning. [BERLACK92]

Although simple in concept, implementing CM can be quite complicated. As multiple versions of the product are produced, tracking the source and object code baseline, documentation, and modules making up the software product escalates. Mainly administrative, CM requires careful execution of a well-defined set of tasks. The procedures for CM must include a predefined strategy for how the product evolution will be managed and facilitated without disrupting the baseline. Careful definitions of baselines and baseline configuration elements are necessary to control the cost and schedule impact of the CM process. Be aware, ***CM can become overburdened if too many baselines are specified and/or too many configuration elements are incorporated into each baseline.*** Another caution is, government and contractor CM teams can become vested in protection of the process and lose sight of their function to manage, control, and accommodate change. If this happens, you must work to promote process improvement within the CM process. [GLASS92]

CM is performed by the development contractor and the program office during the life of the system. It is performed by the contractor during the development process to identify and control evolving versions of software and documentation. **Contractor CM** coordinates activities such as integration of components, testing, repair or modification of the software and documentation. CM is especially useful when multiple users and development organizations are involved. Figure 15-28 illustrates how CM tracks requirements on the F-22 avionics software program.

During development, **Government CM** includes controlling changes to SSS functional and performance requirements. This usually entails chairing and participating on the **Configuration Control Board** that approves requirements changes. The program office is responsible for performing CM (with possible contractor participation) after the software and its documentation are released to the Government. The Government must also assume responsibility for the configuration

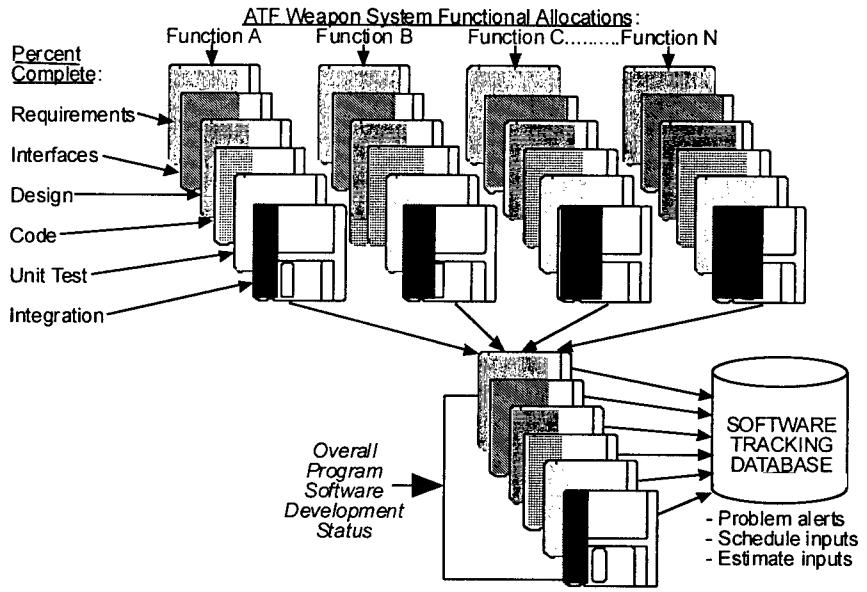


Figure 15-28 Requirements Tracking for the F-22

management of software requirements, implementation, and related deliverables. **Configuration management activities** consist of:

- Identifying and documenting the functional and physical characteristics of each configuration item;
- Controlling changes to each item and its documentation;
- Recording the configuration of actual items; and,
- Auditing each configuration item and its identification. [ROZUM92]

The importance of an efficient, smooth-running CM process cannot be over emphasized. It maintains and improves the evolving software product's quality by preventing the introduction of inconsistencies and defects into the software product and its documentation resulting from the change process. Too often, development teams down-play the CM requirement and view it as a burdensome, *bean-counting* exercise that inhibits rapid response to needed changes. For CM to be effective, it must be properly managed and understood by all team members. It should be used to enhance communication on the status of documents and coding, as well as changes as they are made. It provides a way to identify completed and tested modules that can be reused in subsequent development, and increases software supportability once delivered. Supportability is augmented through

well-defined software elements and a history of the software's development — enabling cost-effective fixes and modifications with little impact on the user. Figure 15-29 illustrates the flow of the overall CM process.

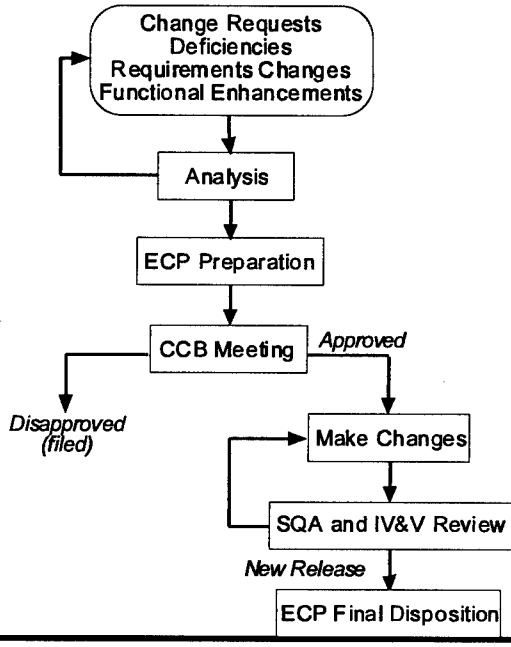


Figure 15-29 Software Configuration Management Process [ROZUM92]

Configuration management also plays an important role in **requirements tracking and control** by ensuring that what was designed (to a specified requirement) is what was built. Because you, the user, the developer, and the maintainer must plan for and analyze many difficult problems simply to define and understand requirements, a good software configuration manager is invaluable. CM captures evolving decisions and written products, and communicates progress and status to you and the development team. A strong CM process is an effective tool for implementing a quality development process. [BERLACK92]

Configuration Management with Ada

Before Ada, the software developer's configuration control, scheduling, change tracking, and other CM activities were usually hidden from, or inaccessible to, the government manager. Configuration information was often obtained through contractor-generated status accounting reports which could be skewed in their favor. An Ada SEE, such as the **ASC/SEE** or **Rational Environment™** *[discussed in Chapter 10, Software Tools]*, has the capability of producing automatic status accounting data for government inspection. This provides greater visibility than was once possible through source code and data management, documentation, test results, and corrective action status reports.

Both government and contractor configuration managers should take advantage of the Ada SEE's CM capabilities. However, managers must have experience (or be trained in) the fundamental concepts of computer programming and in the SEE. They must be able to create/use the tools necessary to develop an Ada program-specific CM system. The contractor's CM plan should describe the Ada SEE and its tools in such a way that ensures data from the SEE can be trusted and used to effectively control the Ada software development. Major Ada CM activities include version control, configuration control, and product release control.

An Ada SEE provides a very efficient CM process that is internal to the software environment. To fully exploit its capabilities, both CM managers must have fundamental Ada language knowledge and object-oriented development (if implemented). They do not have to know how to write Ada code, but must understand Ada's features, such as program units, specifications and bodies, packages, library functions, information hiding, and objects *[all discussed in Chapter 5, Ada: The Enabling Technology]*. [BERLACK92]

> *Progress occurs when courageous, skillful leaders seize the opportunity to change things for the better.*
> — Harry S. Truman

Successful software managers seize every opportunity to improve their software process. Progress occurs when skillful leaders adopt software engineering methods and tools to promote and implement continuous improvement of the procedures and processes that bring quality to their products. User satisfaction, reliability, productivity,

and cost all directly affect the quality of our software. Embracing management techniques that improve the software development process is key to winning the software challenge.

REFERENCES

[AFFOURTIT92] Affourtit, Barba B., "Statistical Process Control Applied to Software," G. G. Schulmeyer and J.I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[ARTHUR93] Arthur, Lowell Jay, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993

[BENDER90] Bender, Gherry, as quoted by Tom Keohler, "On B-1B, Avoiding 'Bugs' Early Beats Debugging Later," *Boeing News*, July 13, 1990

[BOEHM89] Boehm, Barry W., as quoted by Ware Myers, "Software Pivotal to Strategic Defense," *IEEE Computer*, January 1989

[BRYKCZYNSKI93[1]] Brykczynski, Bill and David A. Wheeler, "An Annotated Bibliography on Software Inspections," Institute for Defense Analysis, Alexandria, Virginia, January 1993

[BRYKZYNSKI93[2]] Brykczynski, Bill, et al., "Software Inspections: Eliminating Software Defects," briefing prepared by the Institute for Defense Analysis, February 5, 1993

[CHARETTE89] Charette, Robert N., Software Engineering Risk Analysis and Management, McGrall Hill Book Company, New York, 1989

[CHRUSCICKI93] Chruscicki, Andy, as quoted by Karen D. Schwartz, "Air Force Gets Going with Software Measurement: DecisionVision1 Provides Needed Measurement Tool, But Is It Ahead of Its Time?" *Government Computer News*, September 20, 1993

[COSTELLO88] Costello, Robert B., presentation to the Air Force Scientific Advisory Board, The National Defense University, October 20, 1988

[DEMING82] Deming, W. Edward, Out of Crisis, Massachusetts Institute for Technology, Center for Advanced Engineering Study, Cambridge, Massachusetts, 1982

[DYER90] Dyer, Michael, and A Kouchakdjian, "Correctness Verification: Alternative To Structural Software Testing," *Information and Software Technology*, January/February 1990

[DYER92] Dyer, Michael, The Cleanroom Approach to Quality Software Development, John Wiley and Sons, Inc., New York, 1992

[ESD94] *The Air Force Process Improvement Guide*, Electronic Systems Division, Hanscom AFB, Massachusetts, 1994

[FAGAN86] Fagan, Michael E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July 1986

[FAGAN95] Fagan, Michael E., "Fagan Defect-Free Process: Including the Fagan Inspection Process," briefing presented to Lloyd K. Mosemann, II, The Pentagon, Washington, DC, July 6, 1995

[FISHER91] Fisher, David T., Myths and Methods: A Guide to Software Productivity, Prentice Hall, New York, 1991

[FREEDMAN90] Freedman, Daniel P., and Gerald M. Weinberg, Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products, Dorset House Publishing, New York, New York, 1990

[GILL90] Gill, A., "Setting Up Your Own Group Design Session," *Datamation,* November 15, 1987

[HAUSLER94] Hausler, P.A., R.C. Linger, and C.J. Trammell, "Adopting Cleanroom Software Engineering with a Phased-Approach," *IBM Systems Journal*, Vol. 33, No. 1, March 1994

[HOLDEN92] Holden, James J., "Defect Analysis and TQM," G. G. Schulmeyer and J.I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[HUMPHREY90] Humphrey, Watt S., Managing the Software Process, Software Engineering Institute, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990

[JONES86] Jones, Capers, Programming Productivity, McGraw-Hill Book Co., New York, 1986

[KELLER93] Keller, Ted, briefing "Providing Man-Rated Software for the Space Shuttle," IBM, Houston, Texas, 1993

[KEOHLER90] Keohler, Tom "On B-1B, Avoiding 'Bugs' Early Beats Debugging Later," *Boeing News*, July 13, 1990

[KINDL92] Kindl, LTC Mark R., *Software Quality and Testing: What DoD Can Learn from Commercial Practices*, US Army Institute for Research in Management Information, Communications, and Computer Sciences, Georgia Institute of Technology, Atlanta, Georgia, August 31, 1992

[KRASNER91] Krasner, Herb, "Continuous Software Process Improvement," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[LINGER94] Linger, Richard C., "Cleanroom Process Model," *IEEE Software*, March 1994

[MEYER80] Meyer, GEN Edward C., as quoted in *Military Review*, July 1980

[MONTGOMERY58] Montgomery, Field Marshall Bernard Law, The Memoirs of Field Marshall Montgomery, The World Publishing Co., Cleveland, Ohio, 1958

[MOSEMANN94] Mosemann, Lloyd K., II, comments provided to "Guidelines," May 1994

[O'NEILL94] O'Neill, Don, "Software Inspections Course," white paper, 1994

[PATTON47] Patton, GEN George S., Jr., War As I Knew It, Houghton Mifflin Co., Boston, Massachusetts, 1947

[RAGLAND92] Ragland, Bryce, "Inspections Are Needed Now More Than Ever," *CrossTalk*, November 1992

[RIDGWAY66] Ridgway, GEN Mathew B., "Leadership," *Military Review*, October, 1966

[ROZUM92] Rozum, James A., *Software Measurement Concepts for Acquisition Program Managers*, Technical Report CMU/SEI-92-TR-11/ ESD-TR-92-11, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, June 1992

[RUSSELL91] Russell, Glen W., "Experience with Inspection in Ultralarge-Scale Developments," *IEEE Software*, Vol. 8, No. 1, January 1991

[SET93] "The Cleanroom Software Engineering Process: The Concept and Benefits," briefing prepared by Software Engineering Technology, Inc., Vero Beach, Florida, June 7, 1993

[SONNEMANN94] Sonnemann, Maj R. Michael (USAF), report on study presented to the Software Process Improvement Network meeting, December 7, 1994

[STARS90] Poore, J., et al., "Cleanroom Reliability Manager: A Case Study Using Cleanroom with Box Structures ADL," Software Engineering Technology, Inc., IBM-STARS-CDRL-1949, May 1990

[STARS92] Hart, Hal et al., "STARS Process Concepts Summary," TRI-Ada Conference Proceedings, Orlando, Florida, November 1992

[TIG92] TIG Report, *Functional Management Inspection: Software Independent Verification and Validation (IV&V)*, PN 91-609, The Inspector General of the Air Force, February 6, 1992

[WEISS92] Weiss, Col Daniel H., information provided in correspondence to SAF/AQK regarding February 1992 Guidelines, Directorate of Avionics Management, Headquarters Warner Air Logistics Center, Robins AFB, Georgia, 1992

CHAPTER 15
Addendum A

Improving Software Economics in the Aerospace and Defense Industry

Mike Devlin
Walker Royce

EDITOR'S NOTE

If you have read Chapters 1 through 15, you have traversed virtually the full range of challenges and opportunities associated with software management. These insights, when properly addressed and exploited, will help ensure that you deliver software products on the predicted schedule, at the predicted cost, with the predicted quality and performance desired by the user. On the other hand, by now you might be just a little confused. In seeking a means to bring you an effective summary, the following paper was brought to our attention. Co-written by Mr. Mike Devlin and Mr. Walker Royce of the Rational Software Corporation, this paper succinctly identifies the software engineering practice required to produce more capable defense systems in a more timely and more economic fashion. Although we cannot, and do not, recommend the Rational Software Corporation, we do strongly recommend that you consider them as a benchmark of capability against which to compare competing purveyors of software engineering technology.

INTRODUCTION

Modern aerospace and defense systems incorporate increasingly sophisticated information processing and control systems. These systems contain large amounts of complex software directly in the operational systems themselves, and in the associated development, test, logistics and support systems. This software typically must provide extensive functionality while meeting stringent requirements for safety, security, reliability, availability, and (real-time) performance. The aerospace and defense industry has long recognized that advances in software technology and process improvement are essential to the delivery of more capable systems with shorter development cycles and lower cost.

Rational Software Corporation has been intimately involved in dealing with the pragmatic successes and failures of its customer's software engineering projects across a broad range of Aerospace, Defense, and Commercial applications for over 12 years. The purpose of this paper is to examine three interrelated issues which bear directly on the software capability of the aerospace and defense industry and summarize the current maturity of software engineering practice from Rational's perspective. Advances in *software process*, improvements in *acquisition policy* and a continued *focus on Ada* need to be integrated into a complementary approach to provide breakthrough improvements in the economics of sophisticated software development.

- **Software Engineering in the Aerospace and Defense Industry** examines the state of software engineering in the aerospace and defense industry in comparison to best commercial practice in other industries and defines the elements of a next generation *software process*.
- **The Defense Software Acquisition Process** examines DoD software *acquisition policy* and its impact on the economics of software development in the aerospace and defense industry.
- **Ada and the Aerospace and Defense Industry** examines DoD policy for a continued *focus on Ada* and its impact on the aerospace and defense industry.

The question of the Ada policy has received some recent prominence, influencing the timing and focus of this paper. While Ada is an important issue, it is inappropriate to address the issue of Ada separately from the broader issues of software engineering and

acquisition policy. This paper concludes with recommendations which are based on Rational's experience in employing advanced software technologies in both commercial and defense applications to highlight the discriminating practices of successful projects.

- **Recommendations** presents a set of suggestions for accelerating further adoption of modern techniques within the defense and aerospace industry.

SOFTWARE ENGINEERING IN THE AEROSPACE AND DEFENSE INDUSTRY

Re-engineering the Software Development Process

Over the last decade there has been a significant re-engineering of the software development process, replacing many of the traditional management and technical practices with radically new approaches that combine some hard lessons of experience with advances in software engineering technology. We use the terms ***"object-oriented software process"*** and ***"modern techniques"*** to encompass these new practices. While the essence of this process can be used in most software systems, it is particularly appropriate in situations which are driven by the following needs:

Accommodating change. Those situations where requirements are expected to change over the life of the software, requirements definition requires extensive user input and iteration, or where a flexible architecture is necessary to accommodate growth and change in function, technology, or performance.

Achieving software return on investment (ROI). Those situations where economic considerations require a high degree of reuse of pre-existing components and/or newly developed components within a single system or across multiple systems within a given application domain or line of business.

Value engineering. Projects where there is a need to make accurate, rapid and flexible tradeoffs between cost, schedule, functionality, quality and performance throughout the development process.

Technical or schedule risk. Those situations where schedule pressure (based on mission requirements or time-to-market considerations) or technical uncertainty (complexity, scale, concurrent engineering) require an incremental approach with early delivery of useful versions that provide a solid foundation for further evolution into more complete products over time.

In the commercial world, the combination of competitive pressures, profitability, diversity of customers, and rapidly changing technology cause many systems to have some or all of the above characteristics. In the defense industry it is budget pressures, the dynamic and diverse threat environment, the long operational lifetime of systems, and the predominance of large scale, complex applications which cause many systems to share these characteristics. The paramount need of projects which contain some or all of the above characteristics is one of management control and adaptability. Consequently, our definition of the solution focuses primarily on **process** with strong support from advancing technologies in languages, environments and architectural reuse.

The Elements of an Object-Oriented Software Process

The salient elements of an object-oriented software process include a number of interrelated software engineering practices. We have avoided the use of the terms "*megaprogramming*," "*spiral model*," and "*next generation software process*" even though there is substantial commonality between the techniques of these process frameworks and our presentation. The following themes constitute the recurring practices of successful software projects based on our pragmatic field experience drawn from many sources.

Object-oriented analysis, design, and programming. These techniques replace traditional data-driven methods and functional decomposition methods (structured analysis and design) with an integrated approach to analysis, design and implementation based on an object model.

Rapid prototyping and iterative development. These techniques replace the conventional waterfall model. While there are variations, the basic concept is that early in the development process an initial version of the system is rapidly constructed with an emphasis on addressing high risk areas, stabilizing the basic architecture, and refining the requirements (with extensive user input where possible).

Development then proceeds as a series of iterations building on the core architecture until the desired level of functionality, performance and robustness is achieved. This process places emphasis on the whole system rather than just the individual parts. Through a process of continuous integration, risk is reduced early in the project, avoiding integration surprises late in the project.

Architecture-driven development. Traditionally, the software development process has been requirements-driven, where an attempt is made to provide a precise requirements definition and then implement exactly those requirements. This results in both a process and end products (software) which are very sensitive to even small changes in requirements. In an architecture-driven process the goal is to produce an architecture that is resilient in the face of changing requirements, within some reasonable bounds. The iterative development process then produces a series of architectural prototypes which result in a robust architecture with the required properties.

Large scale reuse. Object-oriented design and an architecture-driven development process implicitly support reuse. However, field experience has demonstrated that reuse must be an explicit management and technical objective in order to achieve economic results. Reuse is most cost effective when reusing reasonably large components (subsystems or class categories), allowing reuse of the analysis, design, integration, and testing of these larger components. Reusing individual classes or modules is important and effective, but has less leverage than reusing larger subsystems consisting of many pre-integrated classes and prefabricated objects.

Software process control and improvement. The transition to an object-oriented software process introduces new challenges and opportunities for management control of concurrent activities and tangible progress and quality assessment. Real world project experience has shown that a highly integrated environment is necessary to both facilitate and enforce management control of the process. An environment that provides semantic integration (where the environment understands the detailed meaning of the development artifacts) and process automation can improve productivity, improve software quality, and accelerate the adoption of modern techniques. For example, it is difficult to fully exploit iterative development if the turnaround time for system builds is measured in days. An environment that supports incremental compilation, automated system builds, and integrated regression testing can provide rapid turnaround for iterative development and allow development teams to iterate more freely.

Software first focus. The onset of open systems standards (e.g., UNIX, TCP/IP), language standards (e.g., Ada, Ada 9X) with highly portable target implementations (e.g., VADS), distributed architecture middleware (e.g., UNAS) and target platform independent development environments (e.g., Rational Apex) has enabled the selection of target technologies (hardware platforms, operating systems, network protocols, and topologies) to be effectively postponed until the optimal time in a project's life cycle. This is crucial to achieving effective software-based tradeoffs between function, performance, cost and schedule in an environment where target technologies are changing dramatically over a project's life cycle.

Each of these elements is related to the others and the combination of the elements is far more powerful than the individual elements. Implementation of these strategies requires a number of organizational and cultural changes as part of re-engineering the development process. As with other paradigm shifts, one must diverge from many of the accepted management practices towards an improved process which better exploits the strengths of new technologies. Resistance to this change is commonplace, especially since it must originate from the senior ranks of project and organizational leaders who are generally comfortable with the status quo.

Relative Maturity of the Aerospace and Defense Software Practices

In order to compare the state of the practice in the aerospace and defense industry with that of commercial industry we examine the question of how the rate of adoption of modern techniques in the aerospace defense industry compares with the rate of adoption in commercial (nondefense) industries.

While object-oriented techniques have received considerable visibility (some would say "*hype*") over the last several years, the aerospace industry has actually been a proving ground for many of its concepts as applied to large systems over the last decade. Some of the early successes which demonstrated the economic benefits of object technology occurred in the defense and aerospace industry (primarily using Ada as the implementation language). The 1991 IDC white paper on object technology (targeted at commercial industry) cites the NobelTech (now CelsiusTech) experience as one of the first demonstrations of the economic payoff from moving to object technology. Beginning in 1986 they used object-oriented design,

Ada and iterative development to achieve large-scale reuse and significantly enhance their competitive position.

Similarly, TRW and the United States Air Force have extensively documented the successes of architecture-driven development on command and control systems. The Command Center Processing and Display System-Replacement (CCPDS-R) project, the Cobra Dane System Modernization (CDSM) project achieved twofold increases in productivity and quality (primarily reductions in delivered error rates and efficiency of software change) along with on-budget, on-schedule deliveries of large mission-critical systems by employing Ada and an iterative development process substantially similar to that described in the previous section. These improvements were largely due to a major reduction in the software scrap and rework (less than 25%) enabled by architecture-driven iterative development, open-minded acquisition practices, and the use of Ada.

While CelsiusTech and TRW were early adopters, over the last four years we have seen momentum shift toward using modern techniques on most of the large aerospace systems where Rational is involved (admittedly a biased sample, since Rational customers tend to be relatively advanced technologically). This shift in momentum represents a fundamental change from the 1983-1987 time frame when Rational first began to recommend this process to customers in the defense and aerospace industry. At that time most programs used functional decomposition, a waterfall life cycle model, requirements-driven development, etc. There was widespread resistance towards moving to these new techniques on a number of fronts.

Program control. Software managers were concerned that iterative development appeared to turn the programmers loose to start coding without requirements or a design. This violated the traditional standard of the waterfall model (*no coding before CDR*) and may have been a valid concern at that time, given that iterative development had not been well formalized and documented. Today, Rational and others have successfully demonstrated iterative development and software technologies for rapid prototyping have matured dramatically. It is now well accepted that iterative development actually gives managers greater control over projects than traditional waterfall models.

Military standards. Program managers were concerned that iterative development was inconsistent with DoD-STD-2167 and other military standards. While many would argue that the military standards did

not define a development methodology, the reality was that the default interpretation and application of the standards did create significant issues. Over time the standards have become more consistent with modern practices, although many government program offices still interpret the standards in a manner which discourages iterative development and incremental deliveries.

Economics. Some early programs did not see the economic case for reuse. Those companies with a large number of fixed-price contracts in competitive markets and those who were interested in producing a reusable product-line immediately saw the benefits and adapted. Those contractors with large cost-plus contracts who felt secure from competition often saw little economic benefit. The current budgetary environment has begun to change attitudes. Program managers more frequently realize that cost and schedule overruns and poor software quality are likely to result in program cancellations in the current environment, rather than creating additional revenue opportunities. Unfortunately, there are still programs today where the resistance to adopting new technology is not based on skepticism that the technology will provide an adequate ROI, but rather concern that the technology will in fact perform as billed, reducing costs and therefore reducing revenue and profits (again this is primarily an issue for cost-plus or level-of-effort contracts).

Inertia. Most program managers are conservative by nature and do not wish to be early adopters of a new technology. This was certainly a reasonable position to take with respect to Ada, object-oriented design and iterative development in the 1983-1987 time frame. Today the inertia is definitely moving in the right direction toward adopting these techniques throughout the aerospace industry.

These obstacles have been (or are being) overcome and the modern techniques of the object-oriented software process described earlier are becoming increasingly common in the aerospace and defense industry. Many large projects (500,000 lines-of-code or greater) have adopted or are adopting these techniques, and many have experienced very positive results. The actual practice (not just study and evaluation) of this next generation software process in aerospace and defense is as widespread as in any other industry segment. This observation is confirmed both by Rational's direct experience with customers and by all of the survey data available from independent research organizations (which Rational purchases as part of its marketing and business planning activities).

Only in the last three years have we seen general acceptance of object technology, architectural focus, and iterative development in other market segments and the usage there is predominantly exploratory rather than full-scale production. Even in the telecommunications industry, an advanced and sophisticated market, the rate of adoption of new techniques is no faster than in the aerospace industry. Three major factors have contributed to the adoption of modern techniques in the aerospace and defense industry.

Leading edge technology. As with many other technologies (semiconductors, materials, algorithms, etc.), aerospace and defense systems have frequently pushed the limits of software technology because of the scale of the systems being built and the extremely demanding requirements. From distributed systems to massively parallel processing, from enormous databases to extreme real-time performance, aerospace and defense systems continually push the limits of what is possible, while also requiring high reliability and affordability. These pressures have demanded the best possible software engineering technology and motivated the exploration, and then adoption of an object-oriented software process.

Focus on engineering rigor. While some market segments have at times emphasized software development as an art and occasionally encouraged a "*hacker*" mentality, the aerospace and defense industry has generally viewed software development as fundamentally an issue of **engineering** discipline. Perhaps because of the deadly serious nature of the defense business, or perhaps because of a similar focus on life-critical software (i.e., commercial avionics and air traffic control systems), the aerospace and defense industry has embraced software engineering as a top priority. This is now true of many other industries (medical instrumentation, telecommunications, etc.) in part because of increasing product liability issues and the focus on total quality management and continuous process improvement. Ironically, this focus on engineering is also largely responsible for producing many of the "*classic*" methods (functional decomposition, waterfall life cycle model, etc.) which sometimes stand in the way of progress.

Transition to Ada. In the late 1970's and early 1980's when Ada was being developed, the primary focus was on producing a single standard language for embedded and mission critical systems, replacing the 400+ languages in use at that time and thereby reducing the tooling, training, development, and maintenance costs associated with DoD software. While Ada did provide that standard language,

an even more important result of the adoption of Ada within DoD has been that Ada has served as a very effective catalyst for the adoption of modern software engineering principles. Some of the early Ada projects did view Ada as "*just another programming language*" like JOVIAL, Fortran, or C. Those projects basically *designed* programs the same way they had in previous languages and simply *coded* them in Ada, achieving few of Ada's benefits while incurring many of the costs of transitioning to a new technology. Fortunately, most of the aerospace and defense industry quickly realized that there was much more to Ada and proceeded to fundamentally re-evaluate all software engineering practices, leading eventually to the adoption of more modern techniques.

On the other side of the ledger, there is one major factor which has inhibited software process improvement in the aerospace and defense industry: the acquisition process.

The Defense Software Acquisition Process

The defense acquisition process and applicable software development standards (e.g., DoD-STD-2167A, MIL-STD-1521B) have historically discouraged the use of iterative development in the defense industry. It is useful to summarize those characteristics of the classic software acquisition process (as it has been typically applied, not necessarily as it was intended) where changes are required in order to enable an object-oriented software process like the one we have described.

Requirements definition. The conventional waterfall model depends upon completely and unambiguously specifying requirements before other development activities, treating all requirements as equally important, and further depends upon those requirements remaining constant over the software development life cycle. These assumptions do not fit the real world. Requirements specification is both the most difficult and the most important part of the software development process. Virtually every major software program suffers from severe difficulties in requirements specification. Moreover, the treatment of all requirements as "*equals*" has drained massive engineering hours away from the driving requirements and wasted those efforts on MIL-STD-required paperwork associated with traceability, testability, logistics support, etc., which is inevitably discarded later as the driving requirements and subsequent design understanding evolve. The intractability of correctly specifying and prioritizing requirements for complex systems has been one of the

primary forces behind the move from the waterfall life cycle model to more iterative life cycle models. Iterative models allow the customer and the developer to work with successive "*prototype*" versions of the system. Pragmatically, requirements can and must be evolved along with an architecture and an evolving set of application increments so that the customer and the developer have a common understanding of the priorities and an objective understanding of some of the cost, schedule and performance tradeoffs associated with those requirements.

Waterfall architecture and design. Conventional techniques also tend to impose a waterfall model on the architecture and design process which inevitably results in late integration and performance showstoppers. In the conventional model the entire system is designed on paper, then implemented all at once, then integrated. Only at the end of this process was it possible to perform system testing to verify that the fundamental architecture (interfaces and structure) was sound. Iterative development produces the architecture first, allowing integration to occur "*as the verification activity*" of the design phase and design flaws to be detected and resolved earlier in the life cycle. This replaces the "*big bang*" integration at the end of a project with continuous integration throughout the project. Iterative development also enables much better quality insight because system characteristics which are largely inherent in the architecture (e.g., performance, fault tolerance, maintainability) are tangible earlier in the process where issues are still correctable without jeopardizing target costs and schedules.

Heterogeneous life cycle format. Given the immature languages and technologies employed in the conventional defense software approach, there was substantial emphasis on perfecting the "*software design*" prior to committing it to the target programming language where it was subsequently difficult to understand or change. This resulted in the use of multiple formats (requirements in English, preliminary design in flowcharts, detailed design in PDL, and implementations in the target language such as Fortran) and error-prone human-intensive translations between formats. The combination of Ada and iterative development enabled a much more homogeneous representation format across the software life cycle, namely a readable, compilable, and executable library of integrated Ada components which eliminated the need for error-prone translations between different, often incompatible formats, in favor of evolutionary refinements in abstraction and ever-increasing depth and breadth of

tangible functionality, quality, and performance. Figure 15-30 illustrates the difference in focus between the intermediate products of the two life cycle models.

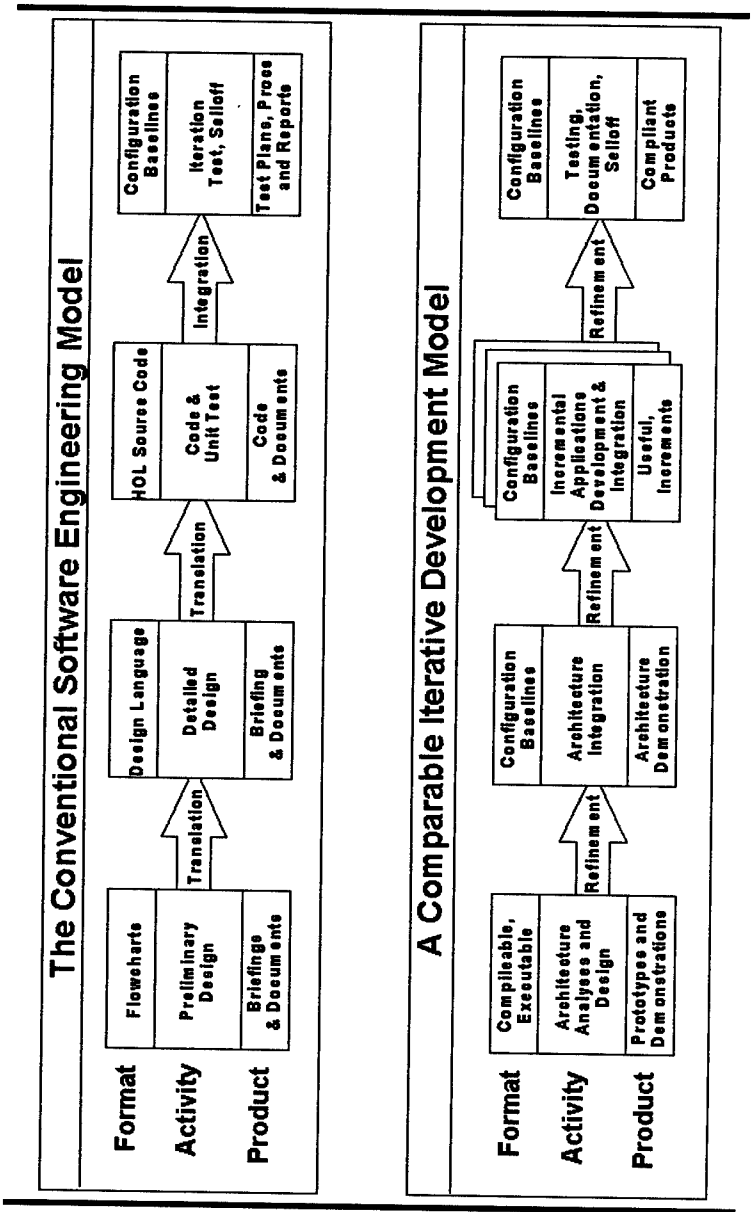


Figure 15-30 Iterative Development Products versus Conventional Development Products

Adversarial relationships. In large part because of the difficulties in requirements specification, the conventional process tends to be adversarial, with the customer and the contractor all too frequently locked in mortal combat. Many aspects of the classic acquisition process degenerate into mutual distrust. This makes it very difficult to achieve a balance between requirements, schedule and cost. A more iterative model, with a closer working relationship between customer, user, and contractor, allows tradeoffs to be made based on a more thorough understanding on both sides. This requires a competent and demanding program office with both application and software expertise and a focus on delivering a usable system (rather than blindly enforcing standards and contract terms) and allowing the contractor to make a profit with good performance. At the same time, it requires a contractor who is focused on achieving customer satisfaction and high product quality in a businesslike manner.

Focus on documents. The conventional process has been focused on producing various documents which attempt to describe the software product, with insufficient focus on producing tangible increments of the products themselves. Major milestones are defined solely in terms of specific documents. Contractors are driven to produce literally tons of paper in order to meet milestones (and get progress payments) rather than spending their energy on tasks that would reduce risk and produce quality software. An iterative process requires actual construction of a sequence of progressively more complete systems which (1) demonstrate the architecture, (2) enable objective requirements negotiations, (3) validate the technical approach, and (4) address key risk resolution. Ideally, both the government program office and the contractor would be focused on these "*real*" milestones with incremental deliveries of useful functionality rather than speculative paper descriptions of the end item vision.

Requirements-driven functional decomposition. A fundamental property of the conventional approach is that it has been very requirements driven with the requirements specified in a functional manner. Built into the classic defense acquisition process is the fundamental assumption that the software itself is decomposed into functional components (CSCIs, CSCs, and CSUs in -2167A terminology), with requirements then allocated to these components. This decomposition is often very different than a decomposition based on object-oriented design and reuse. The functional CSCI decomposition becomes anchored in contracts and subcontracts, often precluding a more architecture-driven approach.

NIH (not-invented-here). The conventional process can discourage reuse between projects and tends to discourage the use of commercial technology. Since requirements are often "*thrown over the wall*" to the software developer, there is little opportunity to negotiate the compromises that are required to reuse an existing product or subsystem. Furthermore, effectively building reusable components or subsystems necessitates investment above and beyond that required for the narrow scope of the project at hand. Ideally the process would encourage customers and contractors to invest in developing reusable architectures which could be applied to a variety of systems in a given domain (avionics, C3I, etc.). Instead, the current process and incentives prevent most investment in reuse through encouragement of specific and singular contract-selfish performance.

Economic incentives. As part of the adversarial nature of the acquisition process, there is considerable focus on ensuring that contractor profits are within a certain acceptable range (typically 5-15%). Occasionally, excellent contractor performance, good value engineering, or significant reuse result in potential contractor profit margins in excess of "*their acceptable initial bid.*" As soon as customers (or their users or government SETA organizations) become aware of such a trend, there is inevitably substantial pressure applied to employ these "*excess*" resources on out-of-scope changes until the margin is back in the acceptable range. As a consequence, the simple profit motive which underlies commercial transactions and incentivizes efficiency is replaced by complex contractual incentives (and producer-consumer conflicts) which are usually suboptimal. Very frequently, contractors see no economic incentive to implement major cost savings, and certainly there is little incentive to take risks which may have a large return. On the other side of the ledger, contractors can easily manage to consume large amounts of money (usually at a small profit margin) without producing results and with very little real accountability for poor performance.

The success of new technologies has led to a more widespread view that the classic defense software acquisition process must be modified or replaced. The new MIL-STD-498, replacing DoD-STD-2167A and DoD-STD-7935A, represents a partial recognition of this problem. The goals of MIL-STD-SDD include removing the implied waterfall model, removing the implied preference for functional decomposition, providing clearer requirements for software reuse, and lessening the emphasis on documents. However, very few of the issues expressed above are dealt with in an explicit manner in the new standard and experience to date has indicated that it is still very difficult to

implement iterative development, since most program offices do not understand the new technologies. Even where there is a relatively advanced program office in favor of using modern practices, the various matrix entities (e.g., IV&V contractors, FFRDC, and SETA contractors, etc.) are wedded to the old process model and are more concerned with protecting their turf (and their jobs) than with producing systems in a more cost effective manner.

Ada AND THE AEROSPACE AND DEFENSE INDUSTRY

Ada is an outgrowth of a remarkable vision that was first enunciated over fifteen years ago. Today, Ada is almost universally recognized as the software industry's premiere language for mission critical software engineering. The struggle to transform the Ada vision into reality (via very useful products) has been pursued with surprising intellectual vigor even though it was far from being the most popular language initiative of the computer science community. Ada's principle *raison d'être* is the DoD's need for a single language in which the software engineering paradigm was supported by, and in some instances enforced within, the semantics of the language. This objective has been very nearly achieved. The table below identifies how the DoD contractor community viewed Ada's risks in 1985 versus risk resolution focus emphasized today. The evolution depicted below represents remarkable progress which is a tribute to DoD and the Ada community.

1985 RISK RESOLUTION FOCUS	1994 RISK RESOLUTION FOCUS
Compiler availability and maturity	Development resource adequacy
Ada language training	O-O and architecture training
Ada software development costs	Reuse costs
Ada environment tool availability	Tool integration and extent of automation
Ada process definition	Iterative development process improvement
Ada/COTS interfaces	Open systems interoperability
Ada runtime overhead	Target resource adequacy

Table 15-6 Ada Risk Evolution from 1985 to 1994

Ada 83's semantics can be characterized as providing strong support for project management functions; somewhat lesser support is provided to the advanced computer science attributes of object-oriented programming which evolved after Ada 83 was baselined. These drawbacks however, will be substantially corrected by Ada 9X. In spite of Ada's success, the invention of new languages has continued unabated in commercial industry. C++ is one example of a relatively new language whose primary design goal was to provide object-oriented programming support (encapsulation, abstraction, polymorphism and inheritance) without compromising the advantages of C (primarily speed and ease of programming). In contrast to Ada, C++ provides little project management support in its selected semantics but it is designed for stronger support to the computer science attributes underlying object-oriented design.

The definition of the Ada language is unique in that it was designed with the goal of enabling better management, design, and architectural control (the higher leverage aspects of software engineering) while sacrificing some of the ease of programming. This is the essence of the Ada culture: top-down control where programmers are subordinates of the lead architects and managers. Other languages, and specifically C++, are focused at simplifying the programming activities while sacrificing some of the ease of control. This of course, is the essence of the C/C++ culture where programmers lead the way. For small programs and noncritical projects, the C++ culture can work well and the Ada culture is perhaps overkill. But for large, complex mission critical systems, the Ada culture is a field-proven necessity for success. Culture is a human-imposed set of trends. Clearly, an Ada culture can be practiced with C++ and vice versa, but the paradigm shift for an organization with cultural inertia is an emotional and extremely difficult undertaking.

It is interesting to note that in the definition of the C++ language, there are many new features which were clearly influenced by earlier Ada advances. Similarly, the O-O features being incorporated in Ada 9X have clearly been motivated by advances in C++. The point here is that both languages have contributed to each other's technical evolution. This language competition has been **healthy** and while there are numerous rhetorical debates about which of Ada or C++ is better, there is very little debate that both of these languages are a quantum leap above all others in supporting the modern techniques of object-oriented software engineering as described within this paper.

CHAPTER 15 Addendum A

As indicated earlier, the single greatest contribution of Ada was to act as a catalyst for the adoption of modern software engineering practices. There has been substantial progress in the software technology in use within this industry over the past decade. Figure 15-31 depicts the on-going transition of software economics from the conventional "*dis-economy of scale*" (caused by the dominance of custom development, *ad hoc* processes and *ad hoc* environments) to the emerging "*megaprogramming*" economy of scale being achieved by organizations who exploit reuse, integrated environments with high levels of automation, and mature, iterative development techniques. Ada, and its associated improvements in environments and process, was to a significant extent, the intermediate catalyst in this transition. In many ways it has been Ada which has turned software engineering into a true professional engineering discipline within this community. Ada provided a truly standard and portable language, widely available on virtually all hardware platforms, with extensive support for modern software engineering principles. Ada has also been a vehicle for introducing new life cycle models, new tools, new design and programming practices, and more secure approaches to the development of high-reliability and safety-critical software. Considerable momentum has been established and this momentum is accelerating with the recent emphasis (in both Ada 83 and Ada 9X) on the use of object-oriented analysis and design with Ada.

Recently there has been considerable discussion of the DoD policy toward Ada, with some polarization between those who believe the current policy should be continued or strengthened and those who believe that the current policy should be abandoned. It is not necessary here to repeat all of the arguments pro and con, but it is useful to examine the key positions and assess their validity from the perspective of Rational's experience. The arguments for continuing the Ada mandate can be reduced to the following:

Technical. Virtually every language evaluation study we know of has concluded that Ada is the best technical language for the DoD domain. Ada has satisfied the goals of the DoD in being a highly reliable and maintainable language. Its strengths include support for large scale projects, ultra-reliable software development, standardization, and real-time support, exactly the needs of the defense domain and other mission critical domains where complexity control and certifiability are required. Ada mandate risks have been substantially resolved whereas the other leading alternative (C++) is faced with many of the same risks that Ada faced 10 years ago (see Table 15-6 below).

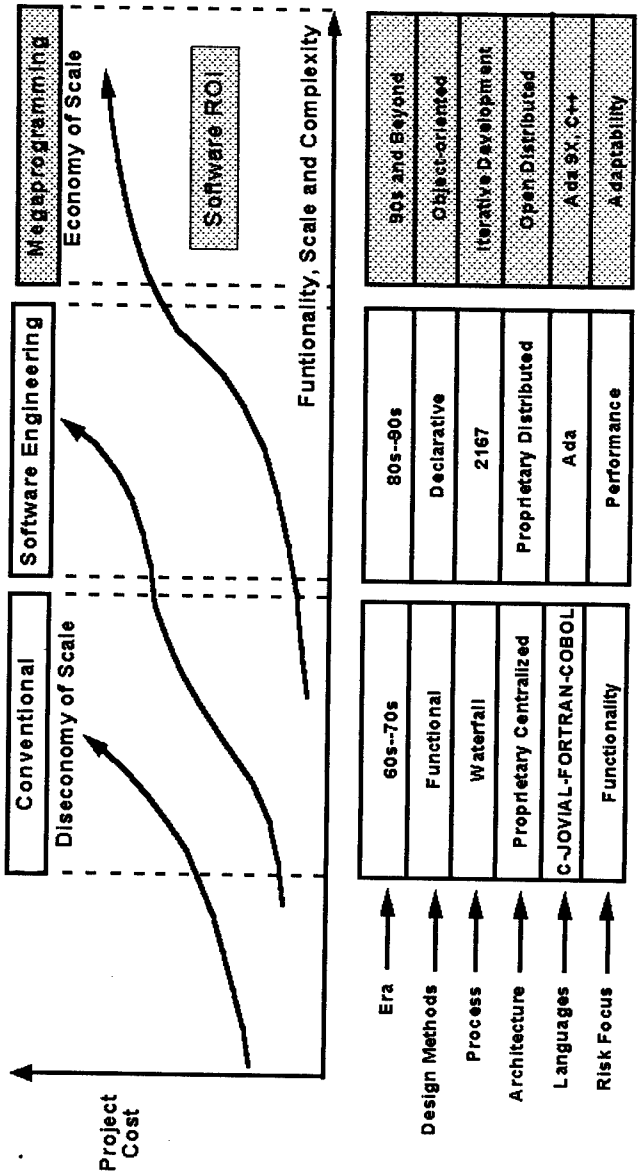


Figure 15-31 Progress Towards Improved Software Economics

Inertia. The past 10 years of DoD investment have resulted in a substantial base of Ada assets including compilers, training, reusable components, and case studies. Furthermore, despite the mandate, Ada is commonly selected by DoD contractor preference and there

are several domains (global air traffic control, NASA, Nuclear Power, FAA, NATO, etc.) that employ Ada in the absence of any mandate.

Standardization. DoD's business case is very different than commercial industry's. The need for a standard language in DoD is motivated by their current practice of organic maintenance with high personnel turnover rates, whereby the costs of tooling and training their maintenance force for a single language have huge economic benefits. This was, in fact, the dominating requirement for Ada's development: to eliminate the divergence of languages, support environments and lack of any ROI in personnel training from assignment to assignment. This need is certainly just as important today as it ever was.

Economic. A substantial number of very large applications (greater than one million source lines) have been successfully delivered and maintained in Ada. While there is certainly not universal success in the financial performance of Ada projects, there is substantial evidence that a mature software organization will perform better with Ada than other languages. There are two important trends of note:

- Prior to Ada, there were (close to) zero large scale projects that delivered on-budget or on-schedule. Over the last 10 years of employing Ada there have been several well-publicized successes.
- Across all projects that have been "*less than successful,*" we know of none that attributed their failure in whole, or in part, to Ada.

The arguments for eliminating the Ada mandate can be summarized as follows:

- **No object-oriented support**. Ada 83, while clearly not supportive of all the object-oriented programming features in vogue today, does support many of the techniques and processes of object-oriented software engineering as described earlier in this paper. In fact, most of the large-scale successful Ada projects Rational is familiar with, were successful predominantly because they were employing object-oriented techniques and modern processes. The primary leverage of these modern techniques is in the process and architecture focus, not in the programming language support. Furthermore, the Ada community has embraced the advantages of object-oriented programming support directly in the language as evidenced by their inclusion in Ada 9X.

- **Lack of commercial support.** The argument that Ada lacks support in the commercial marketplace is subtle. On one hand, anyone that walks into any bookstore's software section will find over 20 books on C++ and maybe 1 on Ada. On the other hand, despite their scarcity in commercial bookstores, there are 30+ textbooks on Ada and it is becoming an increasingly popular vehicle for teaching software engineering at universities. There **are** numerous non-DoD projects who employ Ada for technical and financial reasons. In general, these commercial applications are similar to DoD applications in scale and complexity and the organizations chose Ada for the same reasons as DoD. Perhaps there would be more acceptance of Ada in other commercial applications if DoD had done a better job of marketing, but then again, perhaps not. The real issue here is whether DoD and commercial domain must be closely in synch. A large percentage of the commercial market's software is totally incongruent with DoD software and many commercial practices are equally inappropriate to most DoD software (the glaring exception is DoD's MIS systems which only differ by perhaps their scale). The principal inhibitor of Ada's commercial perception is probably the lack of PC based tools. The impact of PCs on training, software development, and available COTS products is profound. The huge installed base of PCs drives the software market trends and the lack of Ada support on PCs inhibits the single largest source of cheap computing cycles from being part of the Ada solution space. The GNU Ada Translator will help this problem considerably and the emerging next generation PC operating systems will enable today's Ada environments to be more easily transitioned to PC platforms.
- **Insignificant Ada market segment.** This argument is closely related to the previous one but rather than focusing on commercial projects, we examine commercial product markets. In 1992, the Ada compiler and tool market was somewhere in the range of $200-300 million while the C++ compiler and tool market was $300-500 million. While the Ada market is certainly smaller, it is by no means insignificant, and there is ample demand to stimulate considerable investment by small and large companies.
- **Conflict with use of COTS.** There is little debate that there are fewer COTS products available for Ada than there are for some other languages. However, we see no real issue with integrating Ada with COTS products. *Most of the globally important interfaces (DBMSs, GUIs, operating systems, network protocols) have been worked.* Furthermore, there are Ada-based COTS products for development environments (e.g., Apex, VADS), and

architecture middleware (e.g., UNAS) which are better than other language counterparts and provide proven leverage in achieving technical and financial success in large complex software projects.

As implied above, we believe that DoD should **stand firm on the Ada mandate**. In parallel, however, we support the continuing development of both Ada and C++ and DoD should support Ada-C++ interoperability advances which will continue as the C++ language matures (particularly with respect to compiler integrity and language standardization and control) to the current levels of Ada and Ada 95. Ada vendors are already investing aggressively to support this interoperability, C++ vendors should be equally as open. DoD should not consider dropping the Ada mandate; it is an asset in DoD business model which should not be polluted by the "*in vogue*" trends towards commercial practices. Re-evaluating this position and perhaps opening up the mandate to both Ada and C++ after C++ matures into a standard makes good business sense given the rate of software technology advance. However, this maturity level is not likely to occur before 1998. We believe that this long term strategy would promote further investment in Ada-C++ interoperability (which is good for both commercial and DoD domains) and permit some level of healthy competition to continue.

RECOMMENDATIONS

There are several general recommendations that Rational would suggest to policy makers and industry leaders in defense and aerospace. Rational recommends that DoD (and other agencies such as the FAA, NASA, etc.) be more demanding customers with a focus on results. DoD, while not so dominant that it drives the entire software industry, is still a large customer with significant clout. By demanding quality software at reasonable prices on reasonable schedules, DoD can, and will, impact the behavior of the industry. Demanding performance and refusing to tolerate failure will strengthen the industry and encourage the adoption of best practices. Programs with a significant track record of poor management, severe cost and schedule overruns, and poor software quality (defined in terms of fitness for use, not just defects and compliance with narrow specifications) should be terminated promptly, regardless of fault (Government, contractor, whatever). This must be tempered with the understanding that software development contains risk and one must not discourage appropriate risk taking. However, over the medium- and long-term a more businesslike and demanding attitude on the part of the

Government (similar to the commercial market where weak products and producers are eliminated rapidly upon evidence of failure) will be much less expensive than continuing to subsidize poor performance. Without doubt, this is the single most important recommendation we can make.

Rational recommends full and continued support for Ada as a centerpiece of aerospace and defense software policy (DoD, NASA, FAA). As discussed above, we believe there are strong technical and business reasons for using Ada in defense applications.

Rational recommends that DoD continue to encourage the use of commercial-off-the-shelf technology. Procurement and project management practices must consistently encourage use of commercial technology. Today there is insufficient incentive to use commercial technologies and there is absolutely no incentive to compromise often arbitrary requirements in order to allow the use of commercial technology. While some may disagree, we view this as synergistic with the Ada initiative.

Rational recommends continued efforts to streamline and modernize the software acquisition process. The new MIL-STD-498, replacing DoD-STD-2167A and DoD-STD-7935A, is a step in the right direction but does not go far enough with respect to the state-of-the-practice. The pace of such a global change remains excruciatingly slow and most program offices do not understand modern software engineering principles well enough to properly manage software acquisition with the new standard. Further promotion and adoption of iterative development processes (where success and failure signals are more obvious and tangible earlier in the life cycle) is also critical to achieving any kind of success towards our first recommendation. DoD (and the contractor community) must institute a more aggressive program of process improvement to more rapidly evolve the defense software acquisition process into a quality process. DoD must become less insular, reaching out to understand the best practices and lessons-learned in other software markets (more specific recommendations are included below).

Rational recommends stronger support for applied research in software technology in the US Traditionally, software-related technology efforts have been extremely under-invested by both the government and defense contractors. For example, recent awards for the Technology Reinvestment Program were quite discriminatory against funding software related efforts despite the rapidly growing importance

of such technologies. Software technology remains a "*core competency*" of US industry. Not only is advanced software technology developed within the US, but it is rapidly exploited and applied (unlike some other technologies). The US software market remains the largest and the most competitive worldwide and all of the participants (including the defense and aerospace industry) benefit from the dynamic nature of this market. Further investment would maintain this commercial competitiveness as well as benefit DoD software marketplace.

Rational recommends that DoD institute a required training program for all DoD project offices involved in acquisitions with software content greater than some threshold (say $1-5M). This program should be modeled after the Air Force's BOLDSTROKE course but contain more up-to-date project case studies and more focus on software project management and acquisition. Furthermore, while DoD has successfully applied the SEI's Software Capability Evaluations to discriminate contractors with software process maturity, it has yet to apply similar discipline to its own acquisition project offices.

About the Authors

Mike Devlin cofounded Rational in 1981 and served as a member of the Board, Executive Vice President and Chief Technical Officer until he was elected Chairman of the Board in December 1989. Mr. Devlin was appointed Chairman of the Board of Rational Software Corporation and formed from the combination of Rational and Verdix Corporation in March 1994. Mr. Devlin is a graduate of the United States Air Force Academy and was associated with the Air Force Space Division and Satellite Control Facility as a software program manager and as a computer scientist. Mr. Devlin was the Space Division's liaison to the Defense Advanced Research Project Agency on issues relating to modern software languages and methodology. Mr. Devlin graduated first in the class of 1977 at the Academy and was the outstanding graduate in each of his two major fields of study, Engineering and Computer Science. He was awarded a National Science Foundation Graduate Fellowship and received a MS degree in Computer Science from Stanford University in 1978.

Walker Royce is the Director of Software Engineering Process for Rational Software Corporation. Prior to joining Rational Software Corporation, Mr. Royce spent 16 years in a variety of software technology and software management roles at TRW. He was the Project Manager of the Universal Network Architecture Services (UNAS) product-line where he defined and managed its state-of-the-art software process. He served as the Software Chief Engineer responsible for the software process, the foundation Ada components and the software architecture on the CCPDS-R Project, a highly successful, million-line Ada project. Mr. Royce led the development of TRW's Ada Process Model and the UNAS product technologies which have been transitioned from research into practice on numerous large projects and earned him a TRW Technical fellowship and TRW's Chairman's Award for Innovation. His pioneering work in advancing distributed software architecture and evolutionary software process technologies have been published in numerous technical articles and guidebooks and he is a featured lecturer at the Air Force BOLDSTROKE forum on Software Management. Mr. Royce received his B.A. in Physics at the University of California, Berkeley in 1977, MS in Computer Information and Control Engineering at the University of Michigan in 1978, and completed three years of further postgraduate study in Computer Science at UCLA.

CHAPTER 15
Addendum B

Training – Your Competitive Edge in the 90's

Eileen Steets Quann
President, Fastrak Training, Inc.

NOTE: See this article in Volume 2, Appendix O, *Additional Volume 1 Addenda*.

CHAPTER 15
Addendum C

Lessons-Learned from BSY-2's Trenches

Robert F. Sullivan Jr.

NOTE: **See this article in Volume 2, Appendix O, *Additional Volume 1 Addenda.***

CHAPTER

16

The Challenge

CHAPTER OVERVIEW

Success is only to be obtained by simultaneous efforts, directed upon a given point, sustained with constancy, and executed by decision. —Archduke Charles of Austria

In this chapter you will learn that, in addition to detailed technical insight, a high-level, big picture perspective is needed for successful software acquisition management. Closely tied to the technical competence needed for good management is the confidence that you are being supported. From the governing documents, sources for schools and tools, through the white papers and acquisition program examples, to the guidelines and philosophical insights on selected subjects found in these Appendices, you have a wealth of practical information to assimilate and ingest. The Vision for Software expressed here encompasses the promise that you have a software infrastructure to support your management activities. Your challenge is to make use of these resources (e.g., tools, schools, repositories, programs, technology, professional workforce) to ensure the success of your program as it supports the DoD mission.

There are three stages of acquisition management in which all DoD software programs fall. If you are managing a new-start program, your challenge is to follow all the advice found in these Guidelines with the objective of attaining excellence, customer satisfaction, economy, efficiency, and process improvement. If your program is a smooth running on-going effort, your goal is to relentlessly improve your process. This is accomplished through rigorous self-assessment and the introduction of new processes, tools, improved methods, and advanced technologies.

If your on-going program is in trouble, you must first assess the extent of your problems. This assessment is accomplished by defining your process and then quantifying it. You must then establish a measurement baseline and implement a metrics program focused on your problem areas. The cure for a troubled program can only be achieved by identifying the causes of your problems, removing them, and preventing their recurrence. While you are focusing on a cure, there are some bandaid efforts you can employ to get back on track until the sources of problems are identified and remedied. As Benjamin Disraeli, former British prime minister, proclaimed, "He who gains time gains everything." Increasing your schedule will gain you time, productivity, and decrease defects, as will reducing the size of the

software to be developed. If you determine, however, that your program is beyond repair through detailed cost/benefit analyses, do not think twice, ***stop it dead in its tracks!***

Throughout these Guidelines the underlying theme has been ***quality through process improvement****. Your job is never over, nor is your program ever so successful that it cannot be made better. Process improvement means there is a definable, measurable process to improve. The bottom line for improving software development is* ***measurement****. You must be able to determine where you stand today, to determine how to improve for tomorrow. This includes establishing a baseline and measuring progress from that point in time. Measurement should include all facets of your process for which improvement is possible, and for which metrics can be applied as a normal part of everyday activities. Benchmarks are useful for comparing your effort with other successful programs, and for setting realistic goals for improvement.*

These Guidelines are your opportunity for success. They provide you with information you can use to enhance and support your management efforts. You will find no secrets here — only better ways of doing business based on common sense and learning from our mistakes. Remember that success can only be obtained through simultaneous efforts. Your challenge is to take what you have learned here and direct it to your given program. With sustained constancy and sound management decisions, you will help achieve the Vision for Software.

CHAPTER 16

The Challenge

SEIZE THE OPPORTUNITY

In an interview with the *Washington Post*, **General Colin L. Powell** described how to attain success.

> *There are no secrets to success; don't waste your time looking for them...Success is the result of perfection, hard work, learning from failure, loyalty to those for whom you work, and persistence. You must be ready for opportunity when it comes.* [POWELL89]

As a software-intensive system acquisition manager, these Guidelines provide you with a significant opportunity for success. During this critical period of transition for the American military, managers must aggressively look for better ways to increase productivity, reduce costs, and improve product quality. This motivation comes by learning from failure, loyalty to those for whom you work (and who work for you), a persistence to achieve quality through hard work, and a desire for perfection. **Software engineering** is the basis upon which this opportunity resides. The proven paradigms, methods, and tools presented in these Guidelines allow you to take full advantage of this technology.

A software acquisition infrastructure has been established to provide a framework for applying software engineering technology to your program. This infrastructure was designed to be flexible, to take advantage of software state-of-the-art and from management practices that work and will provide you the greatest opportunity for success. However, as **Mosemann** explains,

> *Software problems will not be solved purely by policies, by standards, or even by education. An integrated DoD*

> *software technology strategy that includes both software management and technology initiatives will make a much larger difference in resolving DoD's current and future software problems.* [MOSEMANN93]

Mosemann warns that institutional changes simply do not happen by mandate; there has to be *buy-in* at every level. Your commitment to turn around the software acquisition problems you learned about in Chapter 1, *Software Acquisition Overview*, is ***the most important buy-in of all!*** To do this, all of you who are affected by the infrastructure must participate in its evolution. Incentives must be provided to our industrial partners, along with education and training for our managers, practitioners, and team members. Measurement is an integral part of the framework, as cost/benefits must be understood and quantified. Ways to exploit our valuable cache of legacy software assets through reuse and re-engineering must be explored. Our systems must be open and have well-defined generic architectures so they can evolve and endure. Our customers must be enlightened and our suppliers must be certified. If you are ready for success, ***the opportunity is yours!***

Embrace the Software Vision: Make It Work for You

Although we have turned the tide of failure and experienced some success, we must never be satisfied with the status quo. We must be dedicated to never-ending software process improvement. The **Vision** is to ***continuously improve software quality and predictability through diligent application of engineering discipline***. The way we plan to achieve this Vision is a twofold approach of which you are an integral part. One facet of the Vision encompasses the **institutionalization of software engineering** practice throughout all software development programs DoD-wide. Having read these Guidelines, you have a solid foundation from which to make your contribution to this Vision by institutionalizing the practice of software engineering within your program. Because education and training are key to achieving the Vision, you, as a software manager, must place high priority on keeping your software professionals trained and educated in software engineering discipline.

The other facet of the Vision is the establishment of a **software engineering infrastructure**. As illustrated on Figure 16-1, this infrastructure is based on a concept created by the **Japanese** some 20

years ago — the *house-of-quality*. Used as a **TQM** communication tool, the structure shows how all the pieces of a system are needed to build and provide support to the whole. The importance of the pillars to each other in supporting the ceiling (the Vision) is an inter- and co-related set of methods, techniques, technologies, and organizations. Your side of the equation — using software engineering discipline to build your pillar — needs parallel balance and support from the infrastructure to achieve the Vision for the whole. Here, the purpose is ultimately to help you and other software professionals by actively addressing software issues surfacing within your programs. Part of the infrastructure is the gathering of a software work force within which communication, learning, and education are cultivated and where exchange of corporate knowledge flows freely through technology transfer and the sharing of lessons-learned. Infrastructure resources are dedicated to continuous improvement through working groups and agent (software organizations) support. The infrastructure also brings consistency, repeatability, and currency to software development through the implementation of software policies and management plans.

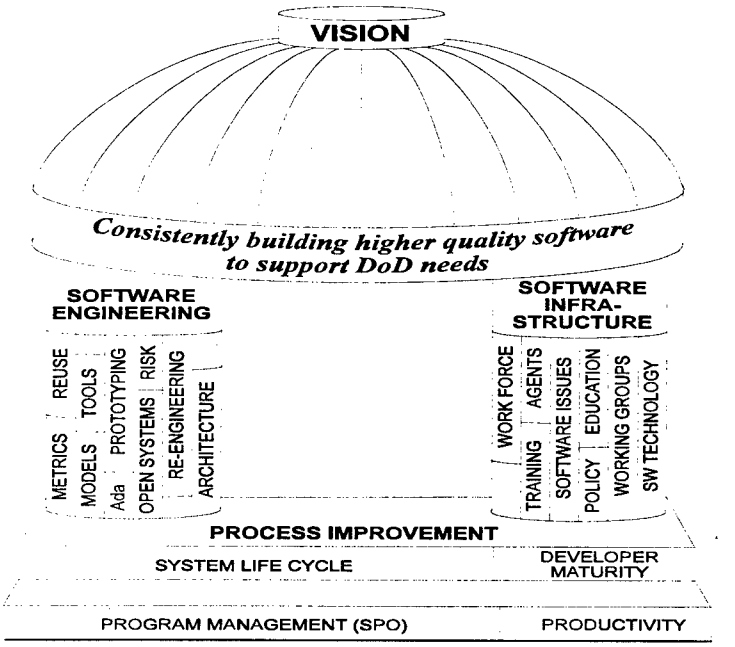


Figure 16-1 Vision for Software

You must realize that the software engineering for which you are responsible is a relatively young discipline. At first it may seem little more than a *hodge podge* of rules, methods, and disparate pieces of information. The Vision provides the unifying theme that brings the ingredients for success into a single software engineering framework. The separate pieces, such as metrics, reuse, models, tools, Ada, prototyping, open systems, re-engineering, risk management, and architecture are interrelated and merged into an integrating foundation permitting us to build quality into our software through the application of technology and practical know-how. This discipline provides an understanding of what it is we are trying to do, and how to go about doing it.

At the foundation of the Vision, holding it all together and making it work, is **process improvement**. The commitment and contribution to this concept must come from your program office, your contractor(s)', your colleagues' programs, and your counterparts within the software infrastructure. The Vision is to select those contractors who have in-hand a predictable, mature, software development process with demonstrable, built-in mechanisms for its continuous improvement. Although this Vision is within our grasp, we must never be satisfied to grab the brass ring. Rather, we must continue polishing it to make it shine, to make it better.

> *Nothing is of greater importance in time of war than knowing how to make the best use of a fair opportunity when it is offered.* [MACHIAVELLI21]

In the heat of fighting your daily management battles, remember the Vision. As you are engineering your software, a software infrastructure provides you the opportunity to do your job better, to help you succeed. This infrastructure is comprised of policies to keep you in tune with initiatives to improve the way we develop our software and manage our acquisitions. DoD and Service policies and instructions are there to make sure we build uniformity and predictability into our systems. Organizations within the infrastructure are there to assist in implementing reuse and metrics, to evaluate our tools and our contractors, and to research new technologies to improve the way we do our jobs. Training programs and software courses provide the opportunity to advance our skills, and to increase our understanding of the software engineering discipline. Make use of the tools, the repositories, the education, the programs, the technology, the agents (labs, institutes, and centers), and the software work force discussed

throughout these Guidelines. They are offered as your fair opportunity; use them to your best advantage. Remember, you are not on a solo mission — a mighty team is there to back you up.

Make the Commitment to Excellence

Embracing the Vision also means making a commitment to excellence. Excellence in management and excellence in your product. ***Excellence in software is zero defects.*** People are conditioned to believe defects in software are inevitable. For the foreseeable future, software will continue to be built by humans; however, humans are believed to have a *built-in defect factor.* Most commercial software development organizations allow 20% of sales for scrap, rework, warranty repairs, complaint handling, service, test, and inspection. [SCHULMEYER92] ***Human errors*** cause this waste. To eliminate waste in software development, we must concentrate on preventing the errors and defects that plague us. There must be a commitment to zero defects for all programs.

In his book, Quality is Free, Cosby explains that a defect which is prevented has no cost. It needs no repair, no examination, no explanation. [COSBY79] Defect prevention techniques *[discussed in Chapter 15, Managing Process Improvement]* can include peer inspections, process action teams, Cleanroom engineering, software quality assurance (SQA), early testing, COTS, reuse, prototyping, and demonstrations. A serious defect prevention program is comprised of combinations of techniques, each chosen for its ability to prevent a different class of defects. ***Remember, zero defects is a state of mind.*** Sun Tzu explained:

> *The principle on which to manage an army is to set up one standard of courage which all must reach.*
> [SUN500BC]

You must set up a ***standard-of-excellence*** which all team members strive to reach. All must commit to the goal of zero defects throughout the development process, and this pledge must be based on a management commitment from you. In some human endeavors we might be willing to accept imperfection. In others, where the success or failure of our mission (or human life) is at stake — defects must be absent!

Although a state of mind, zero defects is not a motivational program. ***Zero defects must be a performance standard.*** Goals and measurable objectives must be part of the process improvement paradigm. Making the management commitment to zero defects in the software you produce is measurable and obtainable. You can accomplish this through sound software engineering discipline and interminable process improvement that progresses towards fewer and fewer defects. **Lt. Gen. Edmonds** expressed this attitude when talking about building software for the F-22.

> *Now is the time for discipline and software. Now is the time for doing it right, not just because of saving money, but for the war fighter — because it's the right thing to do.* [EDMONDS93]

PROGRAM MANAGEMENT CHALLENGE

We are aware that all our readers are not at the same stage in their acquisition programs. The issues with which you are challenged and how you deal with them will, therefore, differ. Your program may be a new start, may be many years into a long acquisition cycle, may be running smoothly, or plagued with the problems cited in Chapter 1, *Software Acquisition Overview.* You might be tasked with the maintenance of newly delivered software, or software that has been in use for 20 years or more built with other-than-Ada code. Or you might be supporting a combination of new Ada software that has to run with older non-Ada applications, or a combination of COTS or NDI. These different management challenges are addressed in the following sections, or in the chapters cited, and are listed as the following:

- Ada-specific challenges encountered with a new-start, on-going, and PDSS program *[see Chapter 5, Ada: The Enabling Technology]*,
- Cleanroom engineering for new-start, on-going, or troubled programs *[see Chapter 15, Managing Process Improvement]*,
- Managing a new-start program,
- Managing an on-going program,
- Managing a PDSS program *[see Chapter 11, Software Support]*, and
- Managing a troubled program.

CHAPTER 16 The Challenge

Managing a New-Start Program

If you are managing a new development, ***follow these Guidelines as completely and fully as possible***. Your challenge is to apply proven software engineering practices and streamlined procurement methods to your acquisition program. They should reflect the concept that ***we are interested in not only buying product, but process***. We have attempted to assemble a variety of lessons-learned experiences on a range of programs to give you insight into what works and what does not. It is also sound advice to research lessons-learned from programs similar to yours within your domain to arm yourself with as much knowledge as is possible. Never forget, software acquisition is one of the toughest management battles you will ever fight. Be armed, prepared, and well-trained. You must always plan, measure, track, and control with quality as your number one goal. Capers Jones expressed this theme when he stated:

> *Quality, reliability, and user satisfaction are factors that separate the leading-edge [organizations] from the laggards. On a global basis, high quality is the main driving force for high-technology products. The [organizations] that recognize this basic fact are poised for success in the twenty-first century; the [ones] that do not recognize it may not last to see the twenty-first century.* [JONES91]

Another major issue to address in your new acquisition is to make sure the new software you are building today is not a maintenance nightmare tomorrow. Well-engineered software must be reliable, understandable, and modifiable. The maintenance burden of tomorrow's legacy software will be lightened by the success of your efforts today.

Managing an On-going Program

Today, there are very few major new-start software-intensive acquisitions in DoD. Therefore, most of the readers of these Guidelines are either managing on-going programs, or programs in PDSS *[discussed in Chapter 11, Software Support]*. If your program is on track, do not be tempted to sit back and rest on your laurels. As Brigadier General Marshall explained:

> *Success is disarming. Tension is the normal state of mind and body in combat. When the tension suddenly relaxes through the winning of the first objective, troops are apt to be pervaded by a sense of extreme well-being and there is apt to ensue laxness in all of its forms and with all of its dangers.* [MARSHALL47]

No one has ever reached and/or maintained a state of perfection in software development. If your program has successfully achieved its first objectives, do not become disarmed by success. There is danger in relaxing your management efforts through a sense of well-being. Your challenge is to relentlessly improve your process through an investment in resources and effort to increase and mature your development capabilities. To improve your process, consider the following axiom.

If It Ain't Broke, Break It!

In an ever-changing, demanding management environment, conventional wisdom says: *"Play it safe; don't question success; and if-it-ain't-broke, don't-fix-it!"* In his book, If It Ain't Broke...Break It!, Robert J. Kriegel, a performance psychology pioneer, explains that the *don't-fix-it syndrome* is very bad advice. [KRIEGEL91] In our highly competitive global environment, following the old axioms of the past could leave you idling in the hangar. To be competitive in the 90's, managers can no longer rely on what worked yesterday. Old thought patterns and definitions of success do not work in a fast-paced, modern world. Even if the past is just 6 months ago, today's rules are different. The plans we made yesterday can be inhibitors to innovation, growth, and maneuvering for the lead. In an interview with the *Washington Post*, **General Schwarzkopf** expressed the unpredictable, chaotic military environment.

> *The analysts write about the war as if it's a ballet...like it's choreographed ahead of time, and when the orchestra strikes up and starts playing, everyone goes out and plays a set piece. What I always say to those folks is, "Yes, it's choreographed and what happens is the orchestra starts playing and some son-of-a-[#*!$!] climbs out of the orchestra pit with a bayonet and starts chasing you around the stage." And the choreography goes right out the window.* [SCHWARZKOPF91]

CHAPTER 16 The Challenge

Who would have predicted just 5 to 10 years ago that the American software community would be playing a desperate game of catchup in an industry we pioneered? As you learned from the *Scientific American* article, "Software's Chronic Crisis," at the beginning of this volume, world-class rivals to the US software machine have their bayonets out, are chasing us around the stage, and are about to take world dominance in the software export industry. The practices of the past that enabled US software producers to achieve international supremacy simply are not working. If we do not throw our old management practices out the window, American software workers will lose their jobs to international competition, similar to what happened to the US automotive, steel, and electronics industries. Table 16-1 illustrates how, although we were once international leaders in software productivity, by 1991 we had started our downward slide.

RANK	MIS SOFTWARE	SYSTEMS SOFTWARE	MILITARY SOFTWARE
1	**United States**	Japan	France
2	France	**United States**	Israel
3	Britain	Germany	S. Korea
4	Canada	France	Britain
5	Switzerland	Britain	Germany
6	Germany	India	Sweden
7	Japan	Taiwan	Italy
8	Norway	S. Korea	**United States**
9	Sweden	Holland	Brazil
10	India	Sweden	Egypt

Source: The Economist, January 23, 1993

Table 16-1 Software Engineering Productivity (1991)

Overseas, software management is more process-oriented, effort-driven, long-term, evolutionary, places importance on the organization's people, and employs slow-growth improvement strategies. In contrast, US software managers are usually results-oriented, schedule-obsessed, performance-driven, short-term, innovative with *giant-leaps-forward* characterized by *big-valleys-after-the-peaks*, place importance on the organization's technological/procedural capabilities, and employ fast-growth strategies. [ZELLS92] Table 16-2 (below) illustrates how

the United States share of the world software market ballooned to approximately 71% in 1982, leveled off to a slow climb of about +4% for the next two years, then took a major plunge of approximately -26% from 1990 to 1993.

YEAR	WORLD SOFTWARE MARKET SIZE (U.S.$)	U.S. SHARE OF WORLD MARKET	% INCREASE/DECREASE U.S. WORLD MARKETSHARE
1982	$51 Billion	71% ($36 Billion) (CBEMA Report, Center for Economic Analysis, Stamfort, CT, 1987)	
1987	$108 Billion	73% ($79 Billion) (CBEMA Report, Center for Economic Analysis, Stamfort, CT, 1987)	
1990	$144 Billion	75% ($108 Billion) (*Industry Week*, January 4, 1993)	
			1982-1990 % Increase **+ 4%**
1993	$277 Billion	49% ($136 Billion) (*Worldwide Information Services Forecast, 1993-98*, INPUT, 1994)	
			1990-1993 % Decrease **- 26%**

Table 16-2 US Share of World's Software and Services Market

NOTE: Although one of the world's fastest growing industries, software is a relatively virgin area in market analysis research. Presently, there are no standardized definitions of market segmentation. Consequently, as explained by the US Department of Commerce, there is a lack of consensus among researchers about software market estimates.

The philosophical management differences between the United States and our world competitors has created a tortoise and hare competition. The US hare has had to slow down, panting, out of breath, while the tortoise methodically plods along passing us up. Actress Helen Hayes expressed why fast rising stars seldom sustain their glow.

> *This is a day of instant genius. Everybody starts at the top, and then has the problem of staying there. Lasting accomplishment, however, is still achieved through a long, slow climb and self-discipline.* [HAYES68]

In the desire for increased productivity and decreased costs, the *harder-faster-longer* mindset simply is not working. The Critical Technologies Update 1994, issued by the **Council on Competitiveness** states that:

> *Although the United States remains at the leading edge of most information technologies, increased competition can be expected in the future... India, for example, is now recognized for its expertise in software... while Taiwan is earning a reputation as a designer and manufacturer of personal computers and peripherals. The United States should closely monitor these and other countries as they build new information technology capabilities.*

This racing, rushing, trying to do more in less time kills quality. People make more mistakes when in a hurry. Performance is poor. There is no time to think of new ideas when you have reached your limits — both physically and mentally. Innovation is nonexistent. Process improvement is impossible. The *harder-faster-longer* mentality simply will not open the gates to the future. As illustrated in Table 16-3, Jones statistics indicate that, as of 1993, the United States ranks third in producing quality software. Of the countries surveyed, we deliver 57% more latent defects in our code than Japan and 15% more bugs than Canada.

RANK	COUNTRY	DELIVERED DEFECTS/ FEATURE POINTS
1	Japan	0.32
2	Canada	0.64
3	**United States**	**0.75**
4	Germany	0.82
5	England	0.83
6	France	0.89
7	Sweden	0.91
8	Norway	0.94
9	South Korea	0.95
10	Netherlands	1.06

Jones (1993)

Table 16-3 Countries with Highest Software Quality

Kriegel says, to be competitive in a global market, we have to break the chains of the past and take an alternate, unconventional route. Unfortunately, managers usually do not change their tactics until they have to. They wait until things are broken and then desperately try to find a *quick-fix*, change strategies, anything to catch up. The problem is, you do not think straight when you are out of fuel, stalled, and in a flat spin. Poor decision making, lack of creativity, and low morale characterizes a development team trying to play catch up. It creates a vicious cycle that keeps them constantly behind.

CAUTION! If, as a program manager (or a software developer), you work 11 to 14 hours every single day, take this as a warning! You are at the ragged edge with no flexibility to meet a real crisis. Stop now and consider whether you need to implement more of the concepts in these Guidelines.

Old habits, doing things the way they have always been done, are major inhibitors to innovation, growth, and progress. You must relentlessly improve your process and your management skills. The time to initiate improvement is not when things are broken, but when they are working well. Kriegel sums up *Break-it Thinking* in unconventional wisdom mixed with common sense. He explains:

- To ride the wave of change, move before the wave hits you.
- Always mess with success.
- Speed kills quality, performance, and innovation.
- The best time to change is when you don't have to.
- Playing it safe is dangerous.
- Get in the habit of breaking your habits.
- Round up your sacred cows and put them out to pasture.
- Stoke the fire, don't soak it; and,
- If it ain't broke, **BREAK IT**! [DRAKE93]

Introducing New Processes, Methods, and Tools

Transitioning a software development program into a mature, software production requires sound management practices, an unremitting obsession for process improvement, and a wise use of technology. Elevating your program's software quality and productivity is neither simple nor cheap, but well worth the investment. New methods can include transitioning to Ada, adding new tools, or altering development methods and practices. As you have learned

throughout these Guidelines, there are many practices, processes, methods, tools, and technologies that offer improvements. These transitions are not always free and may involve some initial schedule and cost impact. You and your contractor(s) should evaluate together the relative merits of the improved practices which seem to offer the greatest potential for reducing overall cost and schedule risk. They must also be assessed for their ability to decrease defects and increase the quality of your product. Software technology transitions are an opportunity for significant gains in quality and productivity, but poorly planned and executed transitions can result in serious program setbacks.

NOTE: See Chapter 10, *Software Tools*, for a discussion on methods, models, tools, and support programs to help you in improving your process.

Successful implementation of ***"new ways of doing business"*** in on-going programs cannot be the exclusive province of either the contractor or the government program manager. Since these *best practices* were not foreseen at contract award, contract documents will not reflect their use and *may* (or may not) need to be modified. Generally, contractors will need to absorb some initial unplanned cost, and the Government will need to concede to some schedule delays. However, if technology transition planning is performed successfully, cost and schedule investments will reap substantial dividends.

The key is to enlighten your customer — educate your contractor — gain a consensus about *"what to do"* and *"how to do it."* ***Be sure they read these Guidelines!*** Take advantage of the infrastructure of support organizations that are doing a lot of the homework for you. They are there to evaluate your needs and advise you on how to proceed. ***Remember the Vision; make it work for you and keep on pressing!***

Managing a PDSS Program

If you are managing a PDSS program, you employ the same tactics as new-start and on-going programs. Follow the software engineering discipline discussed in these Guidelines with the ceaseless goal of improving your process. This can include re-engineering part or all of your code to Ada, incorporating reuse and COTS for enhanced functionality, or restructuring your code so it is more maintainable and modifiable.

Determining If Your Program Is In Trouble

You most likely already know if your program is in trouble! Your developer is not providing orderly documentation, the **SDP** is inadequate, or not being followed. Your program is over budget, behind schedule, and the user-discovered defect rate in delivered modules is above the acceptable range. These are not uncommon problems where a program is on its way to a near disastrous situation. Programs in trouble can run into delays and budget overruns of 200% to 300%, and, in some cases, must be abandoned. [BENNATAN92]

Most software engineering methodologies focus on *preventing* (not *correcting*) these types of problems. Preventing problems is always easier and less costly than solving them. As you have learned throughout these Guidelines, problems become more expensive the further into the development they are discovered. Once neglected, problems propagate into other areas of the development process, making them more difficult and costly to reverse. Your challenge is to determine if your program can be salvaged by enacting a radical change that adopts the ingredients for success found in these Guidelines.

NOTE: If you are not sure whether your program is in trouble, look at C/SCSC (or management metrics) variances. If the current set looks "*abnormal*," you are in trouble!

You must first determine the cause of your program's sickness and the severity of the disease, before you can make a decision about a cure. You must determine whether your program is so sick it should either be terminated, started over from scratch, or whether upgrading your technology and improving your process will provide sufficient remedy. To make this assessment, apply the same software engineering discipline used to prevent problems. ***The best way to identify and assess the severity of your problems is to go looking for them.*** As outlined in Chapter 1, *Software Acquisition Overview*, there are a few basic sources of problems common to most all DoD software programs in trouble. These include:

- Software's inherent complexity,
- Our inability to estimate cost, schedule, and size,
- Unstable requirements, and
- Poor problem-solving/decision making (which includes reliance on *silver bullets*).

Colonel Lyons noted some addition problems:

- Failure to recognize or accept that a software challenge exists,
- Questionable developer capability, capacity, and tools,
- Inadequate development process discipline; and,
- Failure to manage subcontractors. [LYONS91]

Cost, schedule, and quality problems associated with software products are merely symptoms of problems in the process that produced them. Defects, design errors, and major schedule slips are not the causes of problems—they are the ***symptoms***. Behind the symptoms, something was done by someone during the creation or evolution of that activity that caused the problem. By analyzing the cause (e.g., of design errors) and concentrating your resources on the software process, you can determine what must be done to improve that process, and thus, to solve your problems. [ARTHUR93] To determine where in your development process the cause of your problems lie, you have to ***quantify it***. To accomplish this, you must:

- **Define** your process,
- **Measure** your process and product,
- **Analyze** the metrics to determine deficiencies in your process and the quality of your product; and,
- **Institute** the software engineering practices and methods discussed in these Guidelines.

Process improvement implies there is some definable and measurable ***process*** to improve. In software engineering, all processes at each development phase are targets for improvement. There are also ancillary processes, such as configuration management, software quality, test and integration, in-process reviews, and formal peer inspections. Each of these ancillary processes supports your overall development process, each of which can be improved.

To quantify your process, and thus improve it, you must have a ***baseline***. This baseline is used as the measured starting point for each phase of problem solving. You must, therefore, become sufficiently organized to have a definable, quantifiable process that can be measured. [REIFER92] Once measurement data is collected, it must be pondered, analyzed, placed in a larger context, and woven into the fabric of where you have been and where you are going. ***Measurement information must be transformed into "insight" for it to be meaningful.***

What to Do With a Troubled Program

The following **Software Program Managers Network** "***Breathalyzer***" questions will give you a *quick-look* into the status of your program's health. At any time if you cannot answer any of these questions or must answer one or more with a *"no,"* you should schedule an immediate program review.

1. ***Do you have a current, credible activity network supported by a work breakdown structure (WBS)?*** As illustrated on Figure 16-2, an activity network is the primary means to organize and allocate work.

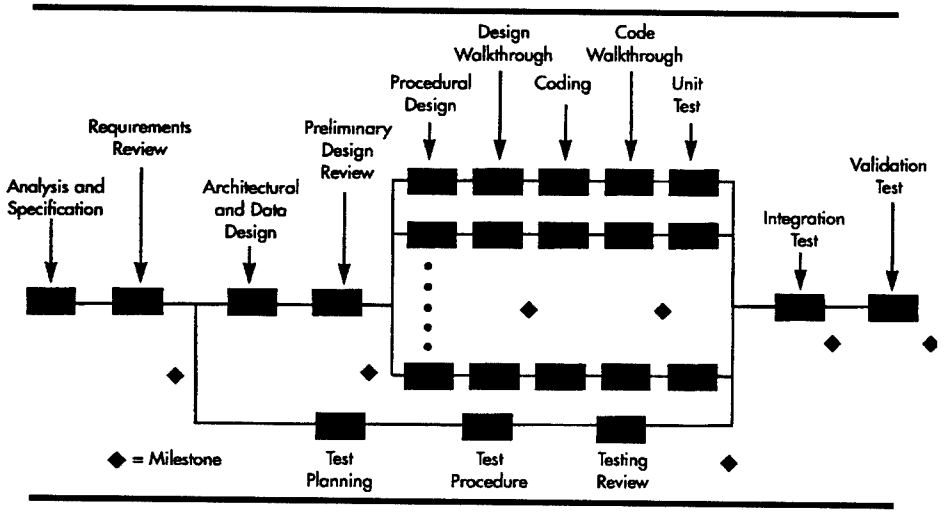


Figure 16-2 Activity Network Example

- Have you identified your critical path items?
- What explicit provisions have you made for work that is not on your WBS?
- Does the activity network clearly organize, define, and graphically display the work to be accomplished?
- Does the top-level activity network graphically define the program from start to finish, including dependencies?
- Does the lowest-level WBS show work packages with measurable tasks of short duration?
- Are program objectives fully supported by lower-level objectives?
- Does each task on the network have a well-defined deliverable?

- Is each work package under budget control (expressed in labor hours, dollars, or other numerical units)?

NOTE: A well constructed activity network is essential for accurate estimates of program time, cost, and personnel needs, because estimates should begin with specific work packages.

2. ***Do you have a current, credible schedule?***
   - Is the schedule based on a program activity network supported by the WBS?
   - Is the schedule based on realistic historical, quantitative performance estimates?
   - Does the schedule provide time for education, holidays, vacations, sick leave, etc.?
   - Does the schedule provide time for quality assurance activities?
   - Does the schedule allow for all interdependencies?
   - Does the schedule account for resource overlap?
   - Is the schedule for the next 3-6 months as detailed as possible?
   - Is the schedule consistently updated at all levels on Gantt, PERT, and critical path charts every two weeks?
   - Is the budget clearly based on the schedule and required resources over time?
   - Can you perform to the schedule and budget?

3. ***Do you know what you have to deliver?***
   - Are system operational requirements clearly specified?
   - Are definitions of what the software must do to support system operational requirements clearly specified?
   - Are system interfaces clearly specified, and, if appropriate, prototyped?
   - Is the selection of software architecture and design method traceable to system operational characteristics?
   - Are descriptions of the system environment and relationships of the software application to the system architecture specified clearly?
   - Are specific development requirements expertly defined?
   - Are specific acceptance and delivery requirements expertly defined?
   - Are user requirements agreed to by joint teams of developers and users?
   - Are system requirements traceable through the software design?

4. ***Do you have a list of your Top Ten risk items? If so, what are they?*** *[See Chapter 6, Risk Management, for more information on the Top Ten List.]*
   - Has a Risk Officer been assigned to the program?
   - Are risks determined through established processes for risk identification, assessment, and mitigation?
   - Is there a database that includes all non-negligible risks in terms of probability, earliest expected visible symptom, and estimated and actual schedule and cost effects?
   - Are all program personnel encouraged to become risk identifiers?
   - Is there an anonymous communications channel for transmitting and receiving bad news?
   - Are correction plans written, followed-up, and reported?
   - Is the database of top-ten risk lists updated regularly?
   - Are transfers of all deliverables/products controlled?
   - Are user requirements reasonably stable?
   - How are risks changing over time?

5. ***Do you know your schedule compression?*** (Schedule compression is an indication of the percent by which this program is expected to outperform the statistical norm for programs of its size and class.)
   - Has the schedule been constructed bottom up from quantitative estimates, not by predetermined end dates?
   - Has the schedule been modified when major modifications in the software take place?
   - Have programmers and test personnel received training in the principal domain area, the hardware, support software, and tools?
   - Have very detailed unit-level and interface design specifications been created for maximum parallel programmer effort?
   - Does the program avoid extreme dependence on specific individuals?
   - Are people working abnormal hours?
   - Do you know the historical schedule compression percentage on similar programs, and the results of those programs?
   - Is any part of the schedule compression based on the use of new technologies?
   - Has the percent of software functionality been decreased in proportion to the percent of schedule compression?

$$ScheduleCompressionPercentage = \left\{1.00 - \left[\frac{CalendarTimeScheduled}{NormalExpectedTime}\right]\right\} \bullet 100$$

(Nominal Expected Time is a function of total effort expressed in person months.)

For example, **Boehm** found that: for a class of DoD programs of 500 person months or more:

$$NominalExpectedTime = 2.15 \bullet [ExpectedPersonMonths]^{.33}$$

(Nominal Expected time was measured from System Requirements Review to System Acceptance Test.) [BOEHM81]

NOTE: Attempts to compress a schedule to less than 80% of its nominal schedule aren't usually successful. New technologies offer additional risk in time and cost.

6. ***What is the estimated size of your software deliverable? How was it derived?***
   - Has the program scope been clearly established?
   - Were measurements from previous programs used as a basis for size estimates?
   - Were source lines-of-code (SLOC) used as a basis for estimates?
   - Were function points used as a basis for estimates?
   - What estimating tools were used?
   - Are the developers who do the estimating experienced in the domain area?
   - Were estimates of program size corroborated by estimate verification?
   - Are estimates regularly updated to reflect software development realities?

NOTE: Software size estimation is a process that should continue as the program proceeds.

7. ***Do you know the percentage of external interfaces that are not under your control?***
   - Has each external interface been identified?
   - Have critical dependencies of each external interface been documented?
   - Has each external interface been ranked based on potential program impact?

- Have procedures been established to monitor external interfaces until the risk is eliminated or substantially reduced?
- Have agreements with the external interface controlling organizations been reached and documented?

8. ***Does your staff have sufficient expertise in the key program domains?***
   - Do you know what the user needs, wants, and expects?
   - Does the staffing plan include a list of the key expertise areas and estimated number of personnel needed?
   - Does most of the program staff have experience with the specific type of system (business, personnel, weapon, etc.) being developed?
   - Does most of the program staff have extensive experience in the software language to be used?
   - Are the developers able to proceed without undue requests for additional time and cost to help resolve technical problems?
   - Do the developers understand their program role and are they committed to its success?
   - Are the developers knowledgeable in domain engineering — the process of choosing the best model for the program and using it throughout design, code, and test?
   - Is there a domain area expert assigned to each domain?

9. ***Have you identified adequate staff to allocate to the scheduled tasks at the scheduled time?***
   - Do you have sufficient staff to support the tasks identified in the activity network?
   - Is the staffing plan based on historical data of level of effort, or staff months on similar programs?
   - Do you have staffing for the current tasks and all the tasks scheduled to occur in the next two months?
   - Have alternative staff buildup approaches been planned?
   - Does the staff buildup rate match the rate at which the program leaders identify unsolved problems?
   - Is there sufficient range and coverage of skills on the program?
   - Is there adequate time allocated for staff vacations, sick leave, training and education?

If you decide, after you have thoroughly analyzed your process and identified the root causes of your problems, that your program is salvageable, you might consider a 3-6 month hiatus to institute the guidance found in this book and get your house in order. By following the software engineering practices discussed here, there is a high

probability you will gain back some or all of the hiatus time you invest in rescuing your program. The Air Traffic Control System in Canada is an excellent example. The program was in trouble. The contractor brought in a new manager whose first action was to ***educate the customer***. Then, it was agreed that a hiatus would occur. It lasted 8 months. During this time many changes were made, including the adoption of the **Rome Laboratory Software Quality Framework**, acquisition of the **UNAS** tool and the **Rational Environment,™** and training of the software development team to a new mindset. As of this writing, the program is on schedule, at cost, and expects to recover most, if not all, of the hiatus time.

In addition, there are some *quick-fix strategies* (as opposed to long-term cures) you can employ if you are truly desperate. If these tactics work, ***you must, must implement software engineering discipline to sustain any permanent improvement.*** Remember, if *quick-fixes* work in the short-term, whatever in your process was causing your problems in the first place must be identified and rectified to sustain long-term improvement. If the root causes are not dealt with, your process will revert back to the problems you identified in your initial process assessment, on an order of magnitude worse. Of course, improving your process is the ideal solution. **Quick-fix strategies** include the following:

- Increase your schedule, and
- Reduce your software size,

Increase Your Schedule

According to **Brooks**,

> *More software programs have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?* [BROOKS75]

When you set your schedule to the minimum development time, effort is at its maximum to meet deadlines, but the number of defects is also correspondingly high. For the troubled (but salvageable) program, the temptation is to throw additional manpower at the problem and hold the schedule. ***This will not work!*** Instead of adding manpower in a desperate attempt to meet unrealistic schedules, extend the development time — without increasing or decreasing manpower. This can substantially reduce the effort (and associated cost) compared

to what it would have taken to accomplish the task on the compressed schedule. In addition, the number of defects will drop. Regrettably, this is often not possible once the program is well underway. If your program is in the 12th month of a 12-month schedule, it is just too late to decide you should have planned in terms of a 17-month schedule. [PUTNAM92] Therefore, the sooner you decide to extend your schedule, the more likely it will be viewed as a credible move by those senior to you.

BEWARE! Adding extra staff to reduce schedule has often not worked. In fact, studies show that it can increase your schedule and increase your defects. Brooks' well-known observation rings true: *"Adding manpower to a late software program makes it later."* [BROOKS75]

The **productivity index (PI)** is a measure of efficiency dependent on variables, such as inherent application complexity and the overall effectiveness of the development organization, where:

$$\frac{Size}{PI} = Effort \bullet Time$$

The **manpower buildup index (MBI)** is an expression of time and effort:

$$MBI = \frac{(TotalEffort)}{Time \cdot \hat{3}}$$

Figure 16-3 illustrates **Putnam**'s minimum-development time concept. The intersection of the *Size/PI* line and the *MBI* line locates the minimum development time. For a 60,000-line MIS developed by an organization with a PI of 15, the minimum development time is 13 months. At this development time, the effort is 78 manmonths with a peak manpower of nine people. To the right of the intersection, marking the minimum development time, lies an area of feasible operating points, two of which are indicated by black dots. By extending the planned development time a few months, you can greatly reduce the effort.

Increasing schedule shrinks the size of the effort with corresponding reductions in defects. Putnam's statistics indicate that a 14-month schedule with a 51-manmonth effort (5 full-time programmers) extended by one month (about 8%) can yield a reduction in effort of 35%. Extending it by 2 months (about 15%) yields a 51% reduction

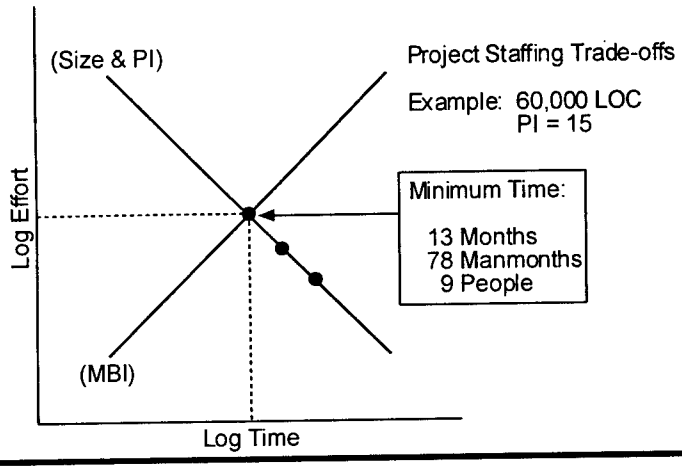


Figure 16-3 Development Time versus Effort Tradeoffs

in effort with corresponding reductions in defects. Thus, the tradeoff of development time for effort can be quite advantageous. [PUTNAM92]

Reduce Your Software Size

There is a distinct correlation between software size, as measured in SLOC or function points, and development time, effort (e.g., manmonths, man-years, cost), manpower, productivity, and the number of defects. (You can determine their precise relationships when measurement data are entered into a database of similar software developments, stratified by the type of application.) If your program is in trouble, reducing the size of your software will reduce development time, effort, the number of defects, and improve programmer productivity. Software size can be reduced by paring the less essential functions from your software, or by deferring the development of separate functions not needed for immediate delivery [i.e., strip the product (with the user's involvement) to the most functions that can be delivered in the time available]. You can also reduce size by cutting the number of newly developed lines-of-code through reuse and implementing **COTS** for generic modules. [PUTNAM92]

Improve Your Process

Improving your process will reduce effort, cost, development time, and the number of defects. ***This is the ideal solution because all management indicators improve.*** Remember, improving your process takes time and should not be considered a *quick-fix*. It takes a long-term strategic commitment. The software development process must be measured for improvements that are both objective and management-oriented. Through measurement, you can determine which are the best strategies to employ for improvement. Choosing a strategy that is, indeed, better will result in software developed in less time, with less effort and money, and increased quality. Improvement requires the ability to answer questions such as:

- When in the software life cycle do errors/defects occur?
- When and how are errors/defects detected?
- What can be done to detect errors/defects earlier?
- When are errors/defects corrected and at what cost?
- What causes and what can prevent the errors/defects that do occur?

Solving software development problems is not just the application of a set of tools, methods, or motivational campaigns. It requires commitment and a dedication to a ***standard-of-excellence***. It is instituting a cultural change, and changing how your team members think and work. ***It involves understanding and enhancing the human process that underlies software development at all levels.*** Improvements can be achieved by changes in procedures, training of personnel, addition of tools, increased automation, and simulated faults insertion. [KENETT92] However, changing the way people think — cultural change — is the greatest challenge, upon which success with process changes depends.

Improvements only occur when rigorous software engineering discipline is applied to improve the human process. The human process must be organized around improvement objectives, properly supported by technology. Whatever it takes to cure your program, ***there must be no turning back to the old ways of doing business!*** DoD has seen its share of software fiascoes. Your challenge is not to let a fiasco turn into a catastrophe, which occurs when we have not learned from our collective mistakes. [REIFER92] There are many techniques and lessons-learned for solving software problems. A few have been introduced here. Others are being discovered daily. Your challenge is to find out what will work for you and implement them! Remember Vince Lombardi's advice,

CHAPTER 16 The Challenge

The greatest accomplishment is not in never falling, but in rising again after you fall. [LOMBARDI68]

What To Do With a Program Catastrophe?

A program catastrophe occurs when the only viable solution is program termination. Examples of circumstances leading to program termination are:

- The program appears to be technically infeasible; i.e., the work cannot be completed given the current state of technology.
- The costs to complete the program far exceed the utility of the final system, or the software will be so costly to operate that the user is better off never implementing it.
- The software will never be completed by a critical date, after which it will not be needed (e.g., an old system will be made to *make-do*).
- The performance quality or maintainability of the software is so bad that the software will be useless when completed — the best way to correct the problem is to start over.
- The software development process is so chaotic, and/or its personnel are so lacking in talent, as to provide no expectation of improvement within a reasonable time, at a reasonable cost.

Abandoning the Catastrophe

If your program is a catastrophe, you must recognize the problem ***as soon as possible!*** The nature of the catastrophe must be identified, and you should treat all efforts and costs expended to date as sunk. This decision is based on a cost/benefit analysis of completing the program, versus restarting it, versus canceling it. Contracting officials should be called in to see if any penalties or restitution to the Government is possible. Sunk costs must be completely disregarded on the common sense principle of ***don't throw good money after bad.*** [ROETZHEIM88]

NOTE: If you have to abandon your program, you should be praised for having the wisdom and fortitude to do so! But, remember, we are all still learning. So by all means, document your lessons-learned and send them to us at the address in the Foreword and last page of this Volume. The benefits of your insights may more than offset present financial losses by helping others to better understand the software management challenge.

THE CONTINUOUS IMPROVEMENT CHALLENGE

> *Standards must be established and maintained in the most routine matters...laxness in these and other routine matters invariably leads to a breakdown in control and discipline...Maintaining high standards requires persistent correction.*
> — Lieutenant General Arthur S. Collins, Jr. (USA)
> [COLLINS78]

As discussed throughout these Guidelines, to achieve continuous improvement you must establish a *software improvement culture* within your program. Everyone on the team (not just the software developers) must be committed to attaining the ***standard-of-excellence*** you set for your program. Because maintaining high standards requires persistent correction, process improvement should be a regular topic of discussion at all in-process reviews and peer inspections. It should also be on the agenda of working group and management meetings held at all levels. Process improvement metrics should be published, discussed, and assessed, the same as budget and schedule status metrics. Your management guidance must support a *"software process first"* philosophy. It is your responsibility to allocate the necessary resources to make improvement happen.

NOTE: See the Addendum to this chapter, "*Reflections on Success*," by Lt. Col. Tom Croak.

Measurement

The most critical factor in the process improvement equation is the collection of metrics. Software **quality metrics** must be collected and analyzed throughout software development. Once you specify a desired ***standard-of-quality*** for each element of importance to your program, achieved levels of quality must be measured at all predefined development milestones. These periodic measures will allow you to assess current quality status, predict the quality level of the final product, and determine where quality is below desired levels. They give you the ability to zero in on problem areas on which process improvement activities can concentrate.

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on how to set up a measurement program.

Baselines

A key element in a measurement program is the **baseline**. It gives you a quantitative view of where you are today. It provides a framework for comparing your development program with historical data, and a context for improvement and innovation. It identifies strengths and weaknesses of the existing process, and helps to communicate them to all stakeholders. [HETZEL93] Baselines are usually established at key milestone points. A meaningful baseline for process improvement must go beyond productivity and quality measures. A complete baseline involves all measurable and improvable facets of the process. These include human resources, organizational structure, user environment, software engineering environment (tools, procedures, technology infrastructure), cost, schedule, funding, management practices — all those things that impact your process. [RUBIN93]

NOTE: Baselines are discussed in Chapter 12, *Planning for Success*.

Benchmarks

Software **benchmarking** is a concept borrowed from the hardware manufacturing industry. Measurements (e.g., failure rates, specifications, time-to-market, cost to produce) are compared with those of competitors. Using these measures, understanding that your production process takes, for instance, 30% more time, costs 20% more, or produces 15% more latent defects than your competitors, makes you realize you are doing something wrong. These figures alone do not tell you what is wrong, they just tell you that you are doing something different that affects your competitive marketplace position.

Benchmarking is a method for establishing baselines by which your development process can be compared and rated against recognized industry leaders. This comparison is used to establish targets and priorities for improving your process to achieve benchmarked levels of performance and quality. [UTZ92]

NOTE: See Chapter 8, *Measurement and Metrics*, for a discussion on the National Software Data and Information Repository (NSDIR), which contains data on software development benchmarks and optimum performance (such as the average number of defect found and corrected in specific software domains).

The quality approach is to fix the process causing the problem rather than fixing the product over and over again. Optimizing your development process can be accomplished by assessing the maturity of your software development capabilities *[discussed in Chapter 7, Software Development Maturity]*. Each time your capabilities are assessed, you will gain insight into those problem areas where you can concentrate your efforts in each subsequent round of process improvement activities. Studies show that process improvement goals continually mature your process, increase quality and productivity, and lower cost. Process improvement and control continues until it is finally time to abandon the process by making a technology transition to a superior process. [UTZ92] One of the most effective ways to transition is to automate all repetitive processes performed by humans. Properly applied, automated tools are one of the best ways to improve the development process. When software professionals are freed from manual labor, they have time to be more creative in their software solutions and process improvement activities.

BEWARE! Studies show that programs operating at low levels of maturity tend to abandon long-term improvement plans when faced with short-term crises. [KRASNER91] **See Chapter 10, *Software Tools*, for a discussion on how to improve your process through automation.**

Quantifiable improvement of software development capabilities requires *buy-in* by all stakeholders in the product and by the owners of all aspects of the process. Improvement activities must be continued and sustained over the entire software life cycle. Improvements should be implemented on all DoD programs in a phased-in, incremental, well-planned manner. Incentives and rewards should be budgeted and granted for improving software capabilities. Your continuous improvement efforts should be sustained until the methods and procedures for improvement become so ingrained in your program's culture that they are performed routinely, as an integral part of every day activities. Remember, *"Success consists in the climb."* [HUBBARD23]

The **Aerospace Industries Association (AIA)** conducts an annual **Quality Assurances Study** that provides an opportunity for participating companies to benchmark their SQA performance with similar companies in the same business for self-examination purposes and to trigger analysis of significant variances. Also, companies participating in the study gain the opportunity to compare their own current year performance with prior periods for self-examination, assessment, and follow-up as appropriate. The focus of the study has been to quantify the SQA function/system in its role in controlling the quality of contractually deliverable products and services. The study concentrates on SQA manpower resources, SQA manpower allocation by function, ratios of SQA manpower to corresponding production labor, and ratios of quality engineering specialties to total quality engineering manpower. Other areas covered by the study include: the contractor's *supplier* SQA cost as a percent of *buy* costs, and the normalization of comparisons of loss measurements which identify rates of in-house *quality losses* (i.e., rework, repair, scrap) to sales (excluding R&D).

Texas Instruments (TI) Benchmarking Process

In 1994, Texas Instruments (TI) conducted a benchmarking study in which nine other major defense companies within their electronics domain participated to exchange performance data, identify industry benchmarks, and establish goals. The study focused on the following key quality metrics:

- On time delivery,
- Production test defects,
- Cycle time reduction, and
- Individual training.

Of the nine companies studied, TI found that:

- Three companies did not measure defects and their reported defect probability range was from 3.5 to 4.0 *sigma [see Chapter 15, Managing Process Improvement, for a definition of "sigma"]*;
- Six companies reported test defects ranging from 3.6 *sigma* to 5.2 *sigma*;
- Three companies had 6.0 *sigma* as a goal (of which two were on track); and
- TI's Defense Software Engineering Group was the Best in Class for an entire company average of 5.23 *sigma*. Figure 16-4 (below) illustrates the results of this benchmarking study. [WILSON94]

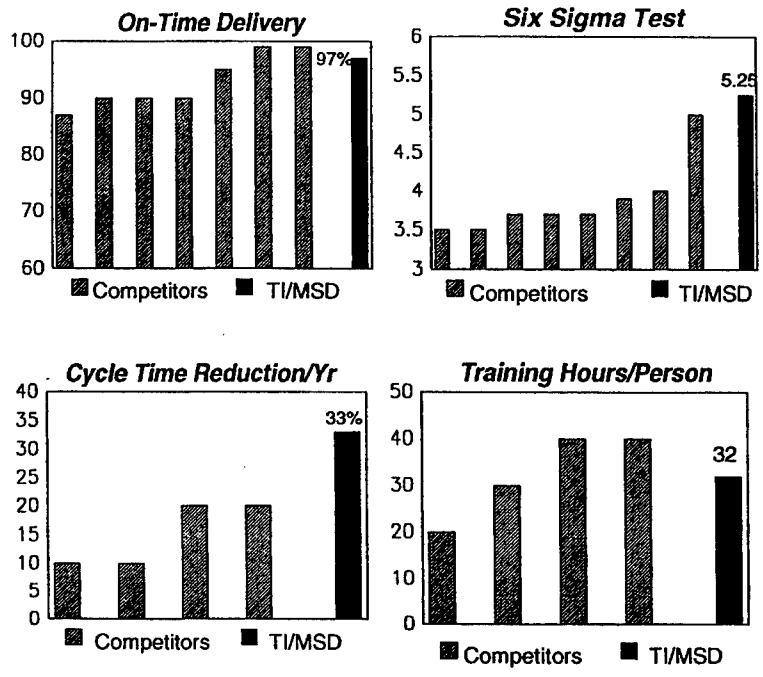


Figure 16-4 TI Quality Metrics Benchmarking Summary (1994 Year to Date) [WILSON94]

YOUR MANAGEMENT CHALLENGE

There are not many true pioneers in software engineering, but few can dispute that **Frederick Brooks** ranks among them. In a now classic collection of essays, Brooks includes a line drawing of a prehistoric tar pit, where great, now extinct creatures are struggling to pull themselves from the gooey abyss. He explains:

> *The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiworks. The management of this complex craft will demand our best use of new languages and systems, our best adoption of proven engineering management methods, liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations.* [BROOKS75]

CHAPTER 16 The Challenge

Your challenge as a software manager is to use the information found in these Guidelines, take control of your acquisition, and develop software with predictable cost, schedule, performance, and quality. In an interview with *Washington Technology*, **Emmett Paige, Jr.**, Assistant Secretary of Defense (C3I), summarized his vision for the future of software development from a DoD perspective.

> *I see a time during my tenure when DoD software engineers use Ada, not just by mandate, but by preference. They have the tools at their fingertips and other processes in place to make a difference. They have open access to quality reusable software components from our reuse repositories and reap the benefits from their controlled use. I see commercial software houses investing in and owning Ada because the market is there, and to do otherwise would be a competitive disadvantage. I see a steady stream of Ada-trained software engineers flowing out of our colleges and universities. I see the Department of Defense viewed as leaders, not followers, in software engineering technology exploitation. I see software costs continuing to go down and quality continuing to go up. And most importantly, I see desire compelling managers to select Ada because of business as well as technical justifications.* [PAIGE93]

Lloyd K. Mosemann, II, Deputy Assistant Secretary of the Air Force for Communications, Computers and Support Systems, has tasked the software community with eight challenges. He remarks that the number *eight* is inadvertently prophetic in that the number eight is the number for new beginnings. There are seven days in a week and on the eighth day you start all over. Your generation of software managers are at a turning point in history as you have the opportunity to start all over with a new order of successful software management. The software community's eight management challenges are:

- To stimulate infrastructure investment,
- To accelerate the pace of technology advance,
- To adopt an architecture mentality,
- To encourage functional managers to become more involved, and to address the fundamentals of how they do their business,
- To advocate technology transition,
- To make greater use of meaningful metrics,

- To reduce the overhead burdens associated with software development, and
- To have defined processes and to institutionalize engineering discipline.

Oliver Cromwell, a famous English statesman and soldier, was on the side of Parliament during the English Civil War. He created the New Model Army (the first professional army in British history), defeated the Scots and the Irish, destroyed the monarchy, executed King Charles I, and ruled England. This illustrious military leader's motto was:

> *Not only strike while the iron is hot, but make it hot by striking.* [CROMWELL47]

The iron is hot! You are equipped with the tools, the repositories, the education, the programs, the technology, the agents (labs, institutes, and centers), and the software infrastructure to help you do your job smarter and better. They are your opportunity to make the iron hot by striking!

ATTENTION! Software Manager, the missile's in your bay! LAUNCH IT!

REFERENCES

[ARTHUR93] Arthur, Lowell Jay, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993

[BENNATAN92] Bennatan, E.M., On Time, Within Budget: Software Project Management Practices and Techniques, QED Publishing Group, Wellesley, Massachusetts, 1992

[BROOKS75] Brooks, Frederick P., Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Reading, Massachusetts, 1975

[COLLINS78] Collins, LTGEN Arthur S., Jr., Common Sense Training, Presidio Press, San Rafael, California, 1978

[COSBY79] Cosby, Philip B., Quality Is Free, New American Library, Inc., New York, 1979

[CROMWELL47] Cromwell, Oliver, Writings and Speeches of Oliver Cromwell, Harvard University Press, Cambridge, Massachusetts, 1947

[DRAKE93] Drake, Dick, review of the book If It Ain't Broke, Break It! by Robert J Kriegel, August 18, 1993

[EDMONDS93] Edmonds, Lt Gen Albert, as quoted by Joyce Endoso, "Ada Gets Credit for F-22 Software Success," *Government Computer Week*, April 26, 1993

[HAYES68] Hayes, Helen, On Reflection, An Autobiography, 1968

[HETZEL93] Hetzel, Bill, Making Software Measurement Work: Building an Effective Measurement Program, QED Publishing Group, Boston, 1993

[HUBBARD23] Hubbard, Elbert, The Roycroft Dictionary and Book of Epigrams, 1923

[JONES91] Jones, Capers, Applied Software Measurement: Assuring Productivity and Quality, McGraw-Hill, New York, 1991

[KENETT92] Kenett, Ron S., "Understanding the Software Process," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[KRIEGEL91] Kriegel, Robert J. and Louis Patler, If It Ain't Broke...BREAK IT! And Other Unconventional Wisdom for a Changing Business World, Warner Books, New York, 1991

[LOMBARDI68] Lombardi, Vince, as quoted by Jerry Kramer, *Instant Replay*, 1968

[LYONS91] Lyons, Lt Col Robert P., Jr., "Acquisition Perspectives: F-22 Advanced Tactical Fighter," briefing presented to Boldstroke Senior Executive Forum on Software Management, October 16, 1991

[MACHIAVELLI21] Machiavelli, Niccolo, from 1521 writings, The Art of War, The Robbs-Merill Co., Inc., Indianapolis, 1965

[MARSHALL47] Marshall, BGEN S.L.A., Men Against Fire, 1947

[MOSEMANN93] Mosemann, Lloyd K., II, as quoted in *Ada Information Clearinghouse Newsletter*, Vol. XI, No. 2, August 1993

[PAIGE93] Paige, Emmett, Jr., "An Endorsement at the Top: DoD Needs to Increase the Appeal of Ada Beyond Its Own Halls," *Washington Technology*, November 4, 1993

[POWELL89] Powell, GEN Colin L., as quoted in the *Washington Post*, January 15, 1989

[PUTNUM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software on Time, Within Budget, Yourdon Press, Englewood Cliffs, New Jersey, 1992

[REIFER92] Reifer, Donald J., "Software Reuse for TQM," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[ROETZHEIM88] Roetzheim, William H., Structured Computer Project Management, Prentice Hall, Englewood Cliffs, New Jersey, 1988

[RUBIN93] Rubin, Howard, "Putting a Measurement Program in Place," Jessica Keyes, ed., Software Engineering Productivity Handbook, Windcrest/McGraw-Hill, New York, 1993

[SCHULMEYER92] Schulmeyer, G. Gordon, "Zero Defect Software Development," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[SCHWARZKOPF91] Schwarzkopf, GEN H. Norman, as quoted in the *Washington Post*, February 5, 1991

[SUN90BC] Sun Tzu, Samuel Grifford, ed., The Art of War, Oxford University Press, New York, 1969

[UTZ92] Utz, Walter J., Jr., Software Technology Transitions: Making the Transition to Software Engineering, Prentise Hall, Englewood Cliffs, New Jersey, 1992

[WILSON94] Wilson, Jesse C., "TI Benchmarking Process," briefing presented to Darleen Druyum, October 4, 1994

[ZELLS92] Zells, Lois, "Learning from Japanese TQM Applications to Software Engineering," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

CHAPTER 16
Addendum

Reflections on Success

Lt Col Tom Croak (USAF)

After winning a Superbowl or the Indianapolis 500, everyone wants to know: "*How did you do it?*" What is the secret of your success? It is as if there were some formula or recipe that can be easily duplicated. The **Air Force STARS Demonstration Project** has been a tremendous success. Now we are getting those questions.

Our success is a direct result of people, process, and technology — focused by desire. If you think about it, the victors of the Superbowl and the Indianapolis 500 would probably have the same answer.

- **People**. You need high quality people who are good at what they do. You could put me in Al Unser's race car, and although I would love to try; I would never have the nerve to get it over 150 mph, about 75 mph lower than qualifying speed, let alone try it with other cars on the track.
- **Process**. An Indi-500 pitstop is the ultimate example of a well-defined process with metrics based continuous process improvement. Fuel and four new tires in 16 seconds!
- **Technology**. There is no doubt that technology is important! Even with the best driver and the best pitcrew, if you put Al Unser in an inferior car, he could finish the race, but would not win.
- **Desire**. Attaining any high goal is only achievable with a strong motivation. Ours was that we sincerely believed we could deliver mission capability into the hands of the warfighters ***Faster, Better, and Cheaper.***

I think you get the picture, you need a quality blend of people, process and technology with a strong desire to succeed. Here is a look at the AF/STARS demo project to see how we developed real command and control systems *Faster, Better, and Cheaper*. Hopefully, there are some lessons-learned that can help you apply STARS technology in your organization.

SUCCESS

The Air Force STARS Demonstration project, also known as the Space Command and Control Architectural Infrastructure (SCAI), has resulted in the creation of a product-line at the Space and Warning Systems Center, Peterson AFB CO, which has produced two command and control systems built in support of Cheyenne Mountain missions. Together, they amount to approximately one MLOC developed in two years, with extremely low defect rates. We have achieved reuse rates of better than 50%.

The demonstrated results of the project have already affected policy on software development. The program's sponsor, Mr. Lloyd K. Mosemann II, Deputy Assistant Secretary of the Air Force for Communications and Computer Support, used our approach as the basic tenants of the new AFI 63-121, which mandates a product-line approach using architecture based domain engineering. We also understand the DoD Software Reuse Initiative has been restructured around product-lines based partially on the results of the Air Force, Army, and Navy STARS Demonstration Projects.

PEOPLE

Software development is a creative process which, simply put, is people centered. The higher the quality of the people, the better the results will be. Quality includes education, training, experience, ability to do abstract thinking, and the ability to communicate ideas. We have been blessed with very high quality people; much higher I believe than average. How do you get these high quality people? It helps if you can show them that you are a forward leaning organization, that is a leader in software engineering, that has the latest technologies, and has developed a nurturing environment that encourages people to try new things.

One of the most remarkable aspects of our success has been the small size of the team relative to the size of the efforts. A well-defined process supported by a Domain Requirements Model, a Domain Logical Model, and Domain reusable assets, all supported by some very powerful code generation tools, allows us to leverage the efforts of a very small group of people. This significantly lowers the number of lines of communication. Small team size also contributes to a level of team building not present in larger groups. Strong team spirit

empowers our walkthroughs to be very productive sessions where the team works together to improve the product.

Another surprise is the ratio of non-programmers to programmers. We have found that the product-line approach shifts more people into support roles. Here is how we are organized: We have a Systems Engineering and Technology support branch which consists of about 12 people performing domain engineering, process engineering, metrics collection and analysis, technology transition and tools support. The Resources Management Branch consists of six people who manage the software engineering environment, perform planning and scheduling, financial management, contracts management, and personnel management. The Product-line branch consists of about 25 people spread across a quality section performing certification and configuration management, and a product section for each of two products currently being produced. Thus a team of only 43 people (16 programmers) was able to build about a million lines of high quality software in about two years.

The "*guilt-free, blame-free*" environment that makes our walkthroughs so effective is a hallmark of our culture, and it has allowed us to truly institutionalize continuous process improvement. If something goes wrong or we get unanticipated results, we focus on the process instead of the person. We always ask the question, "*What was it about our process that allowed the condition to occur?*"

A race car has logos pasted all over it so you know who the contributors are. We have a mix of people from lots of organizations, companies, backgrounds, and experiences and they have been molded into a great team. I would like to thank all the people who have contributed to our success from (alphabetical) Amadeus Software Research, Bishop Engineering, CACI, Canadian Forces, ccPD, CTA, Institute for Defense Analysis, Kaman Sciences, Loral FS, Mantech, PRC, Rational, Robbins-Gioia, SAIC, Software Engineering Institute, SET, Software Technology Support Center, TRW, US Air Force, and our users at USSPACECOM and NORAD. We also had great support from the STARS Center, Electronic Systems Center (ESC/ENS), and our own 21st Space Wing Contracting Squadron.

We believe our strong team spirit is due in large measure to our product-line process and the mix of technologies it integrates. Tools, methods and technology have all contributed to our unusually high productivity — but people are clearly our most important assets.

PROCESS

Before coming to this project, I thought process was just another in a long series of proposed silver bullets that was going to "*save software engineering*." I am now a process believer. My past experience was with monolithic approaches to software development where everyone on the team is working towards achieving the next milestone. In that environment, process can be quite simple and easily understood without tools or people assigned to it full time. In a product-line approach, process becomes a critical component for success. A product-line approach has been compared (rightly so) to a 10 ring circus. There are many simultaneous activities towards multiple milestones. A rigorous approach to process helps keep it all under control.

Our experiences with process, metrics based process improvement, and automated process enactment are noteworthy. With the help of the STARS program and especially the LORAL Federal Systems team, we have pioneered some areas of process- and refined others. These include process capture with the Process Information Organizer Templates we coauthored with the Software Engineering Institute. We have had significant experience using Integrated Computer-Aided Manufacturing (ICAM) Definition (IDEF) to model our processes. We have contributed to the development of both the ProDAT and PEAKS process modeling tools. Our project also contributed to the formulation of the Amadeus metrics collection tool and its use for process improvement. We have experimented with several approaches to automated process enactment.

The big lessons-learned I can pass on to you is that process takes commitment of resources and patience. Process also needs to be built up incrementally. While automated enactment is a fine goal, don't forget to develop simple checklists that people can follow while you are working towards that goal.

In some ways, we have been the victim of our own success. As word of our success spreads, we have lost many people to other projects, both commercial and DoD. We have lost at least six people to MCI which has moved its software operations to Colorado Springs and is trying to implement development approaches similar to ours. I believe it is process that helps us maintain our high productivity rates in spite of turnover approaching 50%.

TECHNOLOGY

We have been using leading edge technology (one person on the team calls it bleeding edge technology) which falls into three groups: process tools, domain modeling tools, and code design and generation tools. These tools support a hybrid methodology developed by combining object-oriented technology and Cleanroom development methods with the incremental build approach called for by Walker Royce's Ada Process Model.

The process tools include ERWin, BPWin, and ProDat for performing IDEF modeling and translating the information into a form that can be used by the PEAKS process modeling tool. In the Domain Engineering area, we use Rational's Rose to capture our systems' characteristics within the framework of our common product-line architecture. We supplement this object-oriented view with an information model view using Cadre's Teamwork.

Our design process essentially refines this domain-level information — using the same two tools. Other design tools are the same architectural infrastructure tools we use to generate much of our code: TRW's UNAS tool to generate the software architecture skeleton, and the Reusable Integrated Command Center (RICC) tools Display Builder and Query Builder, to generate our display and database interfaces. Finally, our Ada development environment is Rational's APEX. All runs on a mixed group of Sun and IBM file servers and workstations.

For more information on the technology used, check out our WWW home page at *http://source.asset.com/stars/loral/scai.html.* The last version of our experience report will be published by the end of the year.

REFLECTIONS

Our partnership with the STARS program has been an excellent one which has proven very beneficial to both sides. It may seem strange to talk about technology last when the prime emphasis of STARS has been on technology development and transition. STARS technology is a key component to our success, but by no means the only one. I felt it was important that the readers clearly understand that you cannot simply take STARS technology and give it to a project and expect them to have the results we did. It takes that blend of people,

process, and technology all focused by desire. You have to really believe that your customers want software built ***Faster, Better, and Cheaper***.

Version 2.0

Acronyms

Blank page.

Acronyms

3GL	third-generation language
4GL	fourth-generation language
5GL	fifth-generation language

A

A&I	analysis and integration
AAM	application architectural model
AAS	Advanced Automated System (FAA)
ABBET	A Broad-Based Environment for Test
ABICS	Ada-Based Integrated Control System
ACC	Air Combat Command
ACEC	Ada Compiler Evaluation Capability
ACM	Association for Computing Machinery
ACT	Advanced Computer Technology (Program) Analysis of Complexity Tool
ACVC	Ada Compiler Validation Capability
ACWP	actual cost of work performed
ADAGE	Avionics Domain Application Generation Environment
AdaIC	Ada Information Clearinghouse
ADARTS	Ada-based Design Approach for Real-Time Systems
ADP	automated data processing
ADPA	American Defense Preparedness Association
AEA	American Electronics Association
AES	Ada Evaluation System
AETC	Air Education and Training Command
AF	Air Force
AFAC	Air Force Advisory Committee
AFAM	Air Force Acquisition Model
AFB	Air Force Base
AFC2S	Air Force Command and Control System
AFCAA	Air Force Cost Analysis Agency
AFCC	Air Force Communications Command
AFCEA	Armed Forces Communications and Electronics Association
AFI	Air Force Instruction
AFIT	Air Force Institute of Technology
AFLIF	Air Force Logistics Information File
AFMC	Air Force Materiel Command
AFMCP	Air Force Materiel Command Pamphlet
AFOTEC	Air Force Operational Test and Evaluation Center
AFP	Air Force Pamphlet (also AFPAM)
AFPAM	Air Force Pamphlet (also AFP)
AFR	Air Force Regulation
AFSC	Air Force Systems Command
AFSCN/CUE	Air Force Satellite Control Network Common User Element

ACRONYMS

AFSPACECOM	Air Force Space Command
AFSPC	Air Force Space Command
AGM 114	HELLFIRE Tactical Missile
AI	artificial intelligence
AIA	Aerospace Industries Association
AIS	automated information system
AITS	adopted information technology standards
AJPO	Ada Joint Program Office
ALC	Air Logistics Center
ALU	arithmetic logic unit
AMRAAM	Advanced Medium Range Air-to-Air Missile
AMSDL	Acquisition Management System Data Requirements Control List
ANSI	American National Standards Institute
APB	acquisition program baseline
API	application program interface
APL	Applied Physics Laboratory
APP	application portability profile
APSE	Ada programming support environment
ARPA	Advanced Research Projects Agency
ASC	Aeronautical Systems Center (USAF)
ASC/SEE	Aeronautical Systems Center/Software Engineering Environment
ASIS	Ada Semantic Interface Specification
ASSET	Asset Source for Software Engineering Technology
ASSIST	Acquisition Streamlining and Standardization Information System
ASST-R	Analytic Software Sizing Tool—Real-time
ATE	automated test equipment
ATF	Advanced Tactical Fighter
AWACS	Airborne Warning and Control System
AWNAM	Automated Weather Network Communications

B

BAA	Broad Agency Announcement
BAFO	best and final offer
BAT	Battlemap Analysis Tool
BCWP	budgeted cost of work performed
BCWS	budgeted cost of work scheduled
BES	budget estimate submission
BLSM	Base-Level Systems Modernization (USAF)
BMD	Ballistic Missile Defense
BPG	Baseline Process Guide
BPR	business process engineering

C

C/SCSC	Cost/Schedule Control System Criteria
C/SSR	Cost/Schedule Status Report
C2	command and control
C3	command, control, and communications
C3I	command, control, communications, and intelligence
C4	command, control, communications, and computers
CAATS	Canadian Automated Air Traffic Control System
CAID	Clear Accountability in Design
CALS	Continuous Acquisition and Life-cycle Support

ACRONYMS

CARD	Cost Analysis Requirements Document
CARDS	Comprehensive Approach for Reusable Defense Software
CASE	computer-aided software engineering
CAST	computer-aided software testing
CBA-IPI	CMM[SM] Base Appraisal — Internal Process Improvement
CBD	Commerce Business Daily
CCA	circuit card assembly
CCPDS-R	Command Center Processing and Display System - Replacement
CD	critical defect
CDA	Central Design Activity (Agency)
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CE	Concept Exploration
	concurrent engineering
CECOM	Communications Electronics Command (Army)
CENTCOM	Central Command
CEO	chief executive officer
CER	cost estimating relationship
CES	Commander's Estimate of the Situation (Navy)
CI	communications interface
	configuration item
CID	commercial item description
CIM	Corporate Information Management
CJCS	Chairman, Joint Chiefs of Staff
CLS	contractor logistics support
CM	configuration management
CMM[SM]	Capability Maturity Model
CMOS	Cargo Management Operations System
CMU/SEI	Carnegie-Mellon University/Software Engineering Institute
CMUP	Conventional Mission Upgrade Program (B-1B Bomber)
CNAM	Conservatoire Nationale des Arts et Metiers
COBOL	Common Business Oriented Language
COCOMO	COnstructive COst MOdel
COE	common operating environment
COFT	Conduct of Fire Trainer
COM	Computer Operations Manual
COMPES	Contingency Operations/Mobility Planning and Execution System
CORBA	Common Object Request Broker Architecture
COTS	commercial-off-the-shelf
CPAF	cost-plus-award-fee
CPAR	Contractor Performance Assessment Report
CPI	cost performance index
CPIF	cost-plus-incentive-fee
CPM	cost performance measure
CPR	Cost Performance Report
CPU	central processing unit
CRCB	Computer Resources Control Board
CRISD	Computer Resources Integrated Support Document
CRLCMP	Computer Resources Life Cycle Management Plan
CRTT	Computer Resource Technology Transition
CRWG	Computer Resources Working Group
CSAD	Computer Systems Authorization Directory

CSC	computer software component
CSCI	computer software configuration item
CSGA	Computer Systems Group Software
CSOM	Computer System Operator's Manual
CSRB	Computer Systems Requirement Board
CSU	computer software unit
CTS	COHESION™ Team/SEE

D

DAA	Defense Auditing Agency
DAB	Defense Acquisition Board
DAC	Designated Acquisition Commander
DAL	data accessions list
DAR	Defense Acquisition Regulations
DCE	distributed computing environment
DDN	Defense Data Network
DDR&E	Director of Defense Research and Engineering
Dem/Val	Demonstration/Validation
DFARS	Defense Federal Acquisition Regulation Supplement
DIA	Defense Intelligence Agency
DID	data item description
DIM	Defense Information Management
DISA	Defense Information Systems Agency
DISN	Defense Information Systems Network
DLA	Defense Logistics Agency
DMA	Defense Material Administration (Netherlands)
DMM	dynamic memory management
DMMIS	Depot Management Information System (DMMIS) (USAF)
DMP	Data Management Plan
DMRD	Defense Management Report Decision
DMSP	Defense Meteorological Satellite Program
DNA	Defense Nuclear Agency
DoD	Department of Defense
DoDD	DoD Directive
DoDI	DoD Instruction
DoDISS	Department of Defense Index of Specifications and Standards
DoDM	DoD Manual
DPA	Delegation of Procurement Authority
DRM	domain requirements model
DS	Database Specification
DSB	Defense Science Board
DSE	Deputy for Software Engineering Deputy for Software Evaluation
DSIC	Defense Standards Improvement Council
DSMC	Defense Systems Management College
DSP	digital signal processor
DSRS	Defense Software Repository System
DSSA	domain-specific software architecture
DT	developmental testing
DT&E	developmental test and evaluation
DTIC	Defense Technical Information Center

ACRONYMS

E

EAC	estimate at completion
ECP	Engineering Change Proposal
ECS	embedded computer system
EDI	electronic data interchange
EDS	Electronic Data Systems
EEI	external environment interface
EIA	Electronics Industries Association
ELSA	Electronic Library Services and Applications
EMD	Engineering and Manufacturing Development
EMX	electronic mobile exchange
EPROM	erasable programmable read-only memory
E-R	entity-relationship
ERD	entity-relationship diagram
ESC	Electronic Systems Center (USAF)
ESD	Electronic Systems Division (now ESC)
ESIP	Embedded Computer Resources Support Improvement Program
ESP	evolutionary spiral process
EUM	End User's Manual

F

FAA	Federal Aviation Administration
FAR	Federal Acquisition Regulation
FATDS	Field Artillery Technical Data System (Army)
FCA	Functional Configuration Audit
FCT	Functional Certification Test
FD	Functional Description
FEA	functional economic analysis
FFP	firm-fixed-price
FFPIF	firm-fixed price incentive fee
FIPS	Federal Information Processing Standard
FIRMR	Federal Information Resources Management Regulation
FMECA	Failure Modes and Effects and Criticality Analysis
FOC	full operational capability
Fortran	Formula Translation for Scientific Applications
FPI	functional process improvement
FPIF	fixed-price-incentive-fee
FQR	Formal Qualification Review Functional Qualification Review
FQT	Formal Qualification Test Functional Qualification Test
FSD	Full Scale Development
FSM	Firmware Support Manual
FTA	fault tree analysis
FY	Fiscal Year

G

GAA	Government Auditing Agency
GAO	General Accounting Office
GCCS	Global Command and Control System
GCSS	Global Combat Support System
GFE	government-furnished equipment

GFI	government-furnished information
GFS	government-furnished software
GKS	Graphics Kernel System
GOSIP	Government Open Systems Interconnect Profile
GOTS	government-off-the-shelf
GPEF	generic package of elementary functions
GPPF	Generic Package of Primitive Functions
GPS	global positioning system
GRMS	generalized rate monotonic scheduling
GSA	General Services Administration
GSBCA	General Services Board of Contract Appeals
GSIS	Graphics Standard Interface Standard
GTE	General Telephone
GTN	Global Transportation Network
GUI	graphical user interface

H

HCI	human-computer interface
HDBK	handbook
HOL	higher-order language
HP	Hewlett-Packard
HQ	headquarters
HSC	Human Systems Center
HTML	HyperText Markup Language
HW	hardware
HWCI	hardware configuration item
HWIL	hardware-in-the-loop

I

I/O	input/output
IASL	Integrated Aircraft Simulation Laboratory
IBM	International Business Machines
ICAM	Integrated Computer-Aided Manufacturing
I-CASE	Integrated-Computer Aided Software Engineering
ICD	Interface Control Document
ICE	independent cost estimate
ID/ATS	Integrated Diagnostic/Automatic Test System
IDA	Institute for Defense Analyses
IDD	Interface Design Description Interface Design Document
IDEF	Integrated Computer-Aided Manufacturing Definition Language
IDL	interface definition language interface design language
IE	information engineering
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IEW	intelligence electronic warfare
IFPP	Instructions for Preparation of Proposals
IFPUG	International Function Point Users Group
ILP	Integrated Logistics Plan
ILS	integrated logistics support
ILSP	Integrated Logistics Support Plan

ACRONYMS

IM	information management
IMCSRS	Installation Materiel Condition Status Reporting System
IMP	Integrated Master Plan
IMS	Integrated Master Schedule
IMTEC	Information Management and Technology Division
IOC	initial operational capability
IOT&E	inital operational test and evaluation
IP	implementation procedure
IPD	integrated product development
IPQM	in-process quality metrics
IPS	information processing system
IPT	integrated product team
IRD	Interface Control Document
IRDS	Information Resource Dictionary System
IRM	information resource management
IRMP	integrated risk management process
IRMS	information resource management software
IRS	Interface Requirements Specification Internal Revenue Service
ISO	International Standards Organization
ISR	information system resources
ISRI	information storage and retrieval interface
ISRS	information system resource software
ITD	Instructions to Defense
ITO	Instructions to Offerors
ITSG	Information Technology Standards Guidance
IV&V	independent verification and validation
IWSM	Integrated Weapon System Management
IWSSF	Integrated Weapon System Support Facility (USAF)

J

JAST	Joint Advanced Strike Technology
J-CALS	Joint-Computer-aided Acquisition and Logistics System
JCS	Joint Chiefs of Staff
JLC	Joint Logistics Commanders
JLSC	Joint Logistics Systems Center
JOVIAL	Jules Own Version of the International Algebraic Language
JSF	Joint Strike Fighter (formerly called JAST)
J-STARS	Joint-Surveillance Target Attack Radar System

K

KLOC	thousand lines-of-code
KPA	key process area
KSLOC	thousand source lines-of-code

L

L(ou)	loss by unsatisfactory outcome
LAN	local area network
LANT	Atlantic
LANTIRN	Low-Altitude Navigating and Targeting Infrared for Night (Operations)
LCM	life cycle management

LCSS	life cycle software support
LCSSE	life cycle software support environment
LOC	line-of-code
LRU	line replaceable unit
LSA	logistics support analysis
LSAR	logistics support analysis record

M

MAA	Mission Area Assessment
MAC	Materiel Acquisition Command Military Airlift Command
MAIS	Major Automated Information System
MAISRC	Major Automated Information System Review Council
MAJCOM	Major Command
MAP	Mission Area Plan
MAPI	Messaging Application Programming Interface (Microsoft)
MAPSE	Minimal Ada Programming Support Environment
MBC	Mortar Ballistics Computer
MBI	manpower buildup index
MCDC	modified condition decision coverage
MDA	Milestone Decision Authority
MDAP	Major Defense Acquisition Program
MICOM	Missile Command (Army)
MIL	military
MilSpec	military specification
MIL-STD	military standard
MilStd	military standard
MIS	management information system
MM	Maintenance Manual
MNA	Mission Needs Analysis
MNS	Mission Need Statement
MOR	memorandum of record
MPR	Mobilization Readiness Priority (Army)
MTA	Maintenance Task Analysis
MTBCF	mean-time-between-critical-failure
MTBF	mean-time-between-failure
MTTD	mean-time-to-defect
MTTF	mean-time-to-failure
MTTR	mean-time-to-repair-and-restore

N

NALCOMIS	Naval Aviation Logistics Command Management Information System
NASA	National Air and Space Administration
NATO	North Atlantic Treaty Organization
NAVMASSO	Naval Management Systems Support Office
NCCOSC	Naval Command and Control Ocean Surveillance Center
NCPII	Fleet Interface for Navy Communications Processing and Routing Re-engineered System
NCSC	National Computer Security Classification
NCTAMS	Naval Computer and Telecommunications Area Master Station LANT

ACRONYMS

NDI	non-developmental item
NDS	non-developmental software
NIPS	Naval Intelligence Processing System
NISMC	Naval Information Systems Management Center
NIST	National Institute of Standards and Technology
NMPPS	Nuclear Mission Planning and Production System
NORAD	North American Aerospace Defense Command
NRaD	NCCOSC RDT&E Division
NSA	National Security Agency
NSDIR	National Software Data and Information Repository
NSF	National Science Foundation
NSIA	National Security Industrial Association
NTDS	Naval Tactical Data System

O

OC-ALC	Oklahoma City - Air Logistics Center (USAF)
OCD	Operational Concept Description
OFP	operational flight program
OMA	organizational maintenance activity
OMB	Object Management Group
	Office of Management and Budget
OO	object-oriented
OOA	object-oriented analysis
OOD	object-oriented design
	object-oriented development
OOP	object-oriented programming
OPR	Office of Primary Responsibility
OPSEC	Operations Security
ORB	Object Request Broker
ORD	Operational Readiness Document
	Operational Requirements Document
ORWG	Operational Requirements Working Group
OSD	Office of the Secretary of Defense
OSE	open system environment
OST	Operational Requirements Document Support Team
OT&E	operational test and evaluation

P

P(ou)	probability of unsatisfactory outcome
P3I	pre-planned product improvement
PAIL	Patrol Aircraft Test Laboratory
PAL	Public Ada Library
PAT	process action team
PB	President's Budget
PC	personal computer
PCA	Physical Configuration Audit
PCMCIA	Personal Computer Memory Card International Association
P-CMM[SM]	People — Capability Maturity Model
PCMS	Process Configuration Management Software
PCO	Project Contracting Officer
PDL	program design language
PDMSS	Programmed Depot Maintenance Scheduling System

PDP	Program Decision Package
PDR	Preliminary Design Review
PDSS	post-deployment software support
PDSSC	Post-Deployment Software Support Concept
PDSSCD	Post-Deployment Software Support Concept Document
PDSSP	Post-Deployment Software Support Plan
PEO	Program Executive Officer
PHIGS	Programmers Hierarchical Interactive Graphics System
PIDS	Prime Item Development Specification
PIF	Productivity Improvement Fund
PIWG	Performance Interface Working Group
PL	Public Law
PMD	Program Management Directive
PMIP	Process Maturity Implementation Project
PMP	Program Management Plan
PMSS	Program Management Support System
POC	point of contact
POE	program office estimate
POM	Program Objective Memorandum
POSIX	Portable Operating System Interface for Unix Computer Environments
PPBS	Planning, Programming and Budgeting System
PRAG	Performance Risk Management Group
PRICE-S	Parametric Review of Information for Costing and Evaluation - Software
PRISM	Portable Reusable Integrated Software Modules
PROM	programmable read-only memory
PRT	Program Requirements Team
PTR	Program Trouble Report
PVI	pilot vehicle interface

Q

QA	quality assurance
QPR	Quantitative Performance Requirement

R

R&D	research and development
RAASP	Reusable Ada Avionics Software Package
RADC	Rome Air Development Center (now Rome Laboratory)
RAH-66	Comanche helicopter
RAP	Risk Aversion Plan
RAPID	Reusable Ada Products for Information Systems Development
RBSE	repository-based software engineering
RCM	requirements correlation matrix
RCS	radar cross section Range Control System
RDBMS	rational database management system
RDT&E	research, development, test and evaluation
RE	risk exposure
REE	Requirements Engineering Environment
RES	risk estimate of the situation
REVIC	Revised Intermediate COCOMO

ACRONYMS

RFI	Request for Information
RFP	Request for Proposal
RICC	Reusable Integrated Command Center
RMA	rate monotonic analysis
RMARTS	Rate Monotonic Analysis for Real-Time Systems
RMP	Risk Management Plan
RMS	reliability, maintainability, and supportability
RO	risk officer
ROADS	Rapid Open Architecture Distribution System (Army)
ROI	return on investment
ROM	read-only memory
RPC	remote procedure call
RPO	Reuse Project Officer
RTE	run-time efficiency run-time environment
RTS	run-time system

S

S&T	science and technology
SAC	Strategic Air Command
SA-CMM[SM]	Software Acquisition — Capability Maturity Model
SAE	Society of Automotive Engineers
SAF	Secretary of the Air Force
SAIC	Science Applications International Corporation
SAME	SQL Ada Module Extension
SAMeDL	SQL Ada Module Description Language
SASET	Software Architecture, Sizing, and Estimating Tool
SBA	standards-based architecture
SBIS	Sustaining Base Information System (USAF)
SCAI	Space Command and Control Architectural Infrastructure
SCE	Software Capability Evaluation
SCN	Specification Change Notice
SCO	Software Change Order
SCP	System Concept Paper
SDCA	Software Development Capability Assessment
SDCCR	Software Development Capability/Capacity Review (now the SDCE)
SDCE	Software Development Capability Evaluation
SDD	Software Design Description Software Design Document
SDF	software development folder
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDK	Sammi Development Kit
SDP	Software Development Plan
SDR	Software Design Review
SDRT	Software Development Risk Taxonomy
SDSA	Software Development and Support Activity
SECDEF	Secretary of Defense
SE-CMM[SM]	Systems Engineering — Capability Maturity Model
SEE	software engineering environment
SEER-M	Software Evaluation and Estimation of Resources - Software Sizing Model

SEER-SEM	Software Evaluation and Estimation of Resources - Software Estimating Model
SEI	Software Engineering Institute (Carnegie-Mellon University)
SEL	Software Engineering Laboratory (NASA)
SEMP	Systems Engineering Management Plan
SEMS	Systems Engineering Master Schedule
SEPG	Software Engineering Process Group
SIDPERS-3	Standard Installation Division Personnel System -Version 3
SIGAda	Special Interest Group on Ada
SISMA	Streamlined Integrated Software Metrics Approach (Army)
SLCSE	Software Life Cycle Support Environment
SLOC	source lines-of-code
SMC	Space and Missile Systems Center
SMM	System Maturity Matrix
SOLE	Society of Logistics Engineers
SOO	Statement of Objectives
SORTS	SOftware Reliability Modeling and Analysis Tool Set
SOW	Statement of Work
SPA	software process assessment
SPAT	software process action team
SPC	Software Productivity Consortium
SPD	System Program Director
	System Program Directorate
SPE	software product evaluation
SPI	schedule performance index
	software process improvement
SPICE	Software Process Improvement Capability dEtermination
SPIP	Software Process Improvement Plan
SPM	Software Programmer's Manual
SPMN	Software Program Managers Network
SPO	System Program Office
SPR	Software Problem Report
SPS	Software Product Specification
SQA	software quality assurance
SQL	Structured Query Language
SQM	software quality management
	software quality maturity
SQPP	Software Quality Program Plan
SrA	Senior Airman
SRAMII	Short-Range Attack Missile II
SRC	Software Re-engineering Center
SRE	Software Risk Evaluation
SRI	Software Reuse Initiative
SRS	Software Requirements Specification
	System Requirements Specification
SSA	Software Support Agency (Activity)
	Source Selection Authority
SSC	Standard Systems Center
SSDD	System/Segment Design Description
	System/Segment Design Document
SSE	Source Selection Evaluation
SSEB	Source Selection Evaluation Board

ACRONYMS

SSE-CMM[SM]	Systems Security Engineering — Capability Maturity Model
SSET	Source Selection Evaluation Team
SSN 21	USS Seawolf Submarine
SSO	System Security Officer
SSP	Source Selection Plan
SSPM	Software Standards and Procedures Manual
SSRT	Single-Stage Rocket Technology
SSS	System/Segment Specification
	System/Subsystem Specification
STAMIS	Standard Army Management Information Systems
STANFINS-R	Standard Finance System - Redesign (Army)
STAR	System Threat Assessment Report
STARS	Software Technology for Adaptable, Reliable Systems
STC	Software Technology Conference
STD	Software Test Description
	Software Test Document
	standard
STE	software test environment
STEP	Software Test and Evaluation Panel (Army)
STP	Software Test Plan
	software test procedure
	System Test Plan
STR	Software Test Report
	Software Trouble Report
STSC	Software Technology Support Center
SUM	Software User's Manual
SUS	Software Unit Specification
SWCI	software configuration item
SWSC	Space and Warning Systems Center (USAF)

T

T&E	test and evaluation
TAC	Tactical Air Command
TAFIM	Technical Architecture Framework for Information Management
TAR	Test Analysis Report
TBQ	Taxonomy-based Questionnaire
TCP/IP	Transmission Control Protocol/Interconnect Protocol
TCRMS	Type Commander Readiness Management System
TCSEC	trusted computer system evaluation criteria
TCTO	time-compliance technical orders
TDT	Theater Display Terminal
TDY	temporary duty
TEMP	Test and Evaluation Master Plan
THAIS	Type Commander's Headquarters Automated Information System
TIM	Technical Interchange Meeting
TMD	tactical munitions dispenser
TMP	Technical Management Plan
TP	Test Plan
TPM	technical performance measurement
TPS	test program set
TQAE	test quality assurance evaluator
TQM	Total Quality Management

TQMP	Total Quality Management Plan
TRM	Team Risk Management technical reference model
TRMS	TYCOM Readiness Management System
T-SCE	Trusted Software Capability Evaluation
TS-CMM℠	Trusted Software — Capability Maturity Model
TWG	technical working group

U

UM	User's Manual
UNAS	Universal Network Architecture Services
USAISC	US Army Information Systems Center
USAISSDCL	US Army Information Systems Software Development Center - Lee
USD/A	Under Secretary of Defense for Acquisition
USSTRATCOM	US Strategic Command
UVPROM	ultraviolet PROM
UX	Unix

V

VCC	VHSIC central computer
VDD	Version Description Document
VHDL	VHSIC Hardware Description Language
VHLL	very high-level language
VHSIC	very high-speed integrated circuit
VSCS	Voice Switching and Control System

W

WBS	work breakdown structure
WMMP	Woman Marine Model Prototype
WPFA	War Planning Systems Directorate
WR-ALC	Warner Robbins Air Logistics Center (USAF)
WSRD	Worldwide Software Resource Directory
WWMCCS	Worldwide Military Command and Control System
WWSS	Warfare and Warfare Support System

Index

Blank page.

Index

1553 databus. 5-70, 10-71
1912th Computer Systems Group. 5-67

A

Abstraction. 4-34, 14-25, 14-31
Acquisition. 1-44
  Acquisition Plan. 9-37
  Baseline. 1-17
  Cleanroom-based. 15-56
  Fast track. 3-22
  Plan. 5-65, 12-12, 12-42
  Program baseline (APB). 12-46
  Reform. 2-17
  Reform working group. 2-17
  Strategy. 3-7, 12-12, 12-47
    best-value. 13-36
    design-to-cost. 12-18
    event-based contracting. 12-47
    panel. 9-37
    process-driven. 1-46
    schedule-plus. 13-9
    System Concept Paper (SCP). 12-13
  Streamlining. 2-14
Acquisition Streamlining and Standardization Information System (ASSIST). 2-22
Ada. 11-12, 14-67, 15-58
  Ada 95. 5-5, 10-71
    Project Office. 5-59
    transitioning to. 5-62
  Commercial market. 5-5
  Compiler. *See Compiler*
  Component interfaces. 5-9
  Cost of. 5-39, 5-41, 8-49
  Documentation. 14-79
  Experience. 8-43
  F-22. 11-12
  Fault rate. 5-41
  Implementation. 5-62
    mixing with other languages. 5-76
    porting to. 5-76
  In the RFP. 5-78
  Interface standards. *See Standards*
  Language features. 5-38
    body. 5-51
    exception handling. 5-54
    generics. 5-55, 11-12
    input/output package. 5-56
    interference. 5-11
    package. 5-11, 5-31, 5-35, 5-52, 5-70
    pointer. 5-60
    primitive. 5-60
    program unit. 5-51
    representation specification. 5-56
    specification. 5-51, 14-38
    subprogram. 5-52
    tasks. 5-53
    typing. 5-35, 5-60
  MIS systems. 5-6
  Policy. 5-50
  Process Model. 5-78, 15-53
  Public Ada Library (PAL). 10-72
  Re-engineering to. 11-17
  Requirements. 5-77
  Reuse. *See Reuse*
  Run-time efficiency. 5-73
  Standardization. 2-24, 5-7
  Technology. 5-65
    Insertion Program. 10-71
  Use. 15-59
  Waiver. 2-22
  Windows Project. 5-31
Ada and C++ Business Case Analysis. 5-37
Ada Information Clearinghouse (AdaIC). 10-71
Ada Language Reference Manual. 10-35
Ada Quality and Style Guidelines. 10-33
Ada Semantic Interface Specification (ASIS). 5-71, 10-73
Ada Technology Insertion Program. 10-71
Ada-ASSURED. 10-33
Ada-based Design Approach for Real-time Systems (ADARTS). 10-45
Ada-based Integrated Control System (ABICS). 5-3
AdaMAT. 10-66

AdaQuest. 10-58, 10-66
ADARTS℠. 15-62
AdaSAGE. 10-49
AdaTEST. 10-60
AdaWISE. 10-59
Adopted Information Technology Standards (AITS). *See Standards*
Advanced Automation System (AAS). 12-14
Advanced Computer Technology (ACT) Program. 10-74
Advanced Medium-Range Air-to-Air Missile (AMRAAM). 15-12
Advanced Research Projects Agency (ARPA). 9-21, 9-27, 9-34, 10-77
Aeronautical Systems Center/ Software Engineering Environment (ASC/ SEE). 10-28
  ASC/SEE. 15-71
Aerospace Industries Association (AIA). 2-17, 16-31
AF Sup 1 to DoDI 5000.2. 5-79
AFI 10-601. 3-6
AFMCP 63-103. 7-7
AFMCP 64-102. 13-11
Aggregate. 14-42
AH-64 Apache. 2-4, 9-3
Air Force Acquisition Model (AFAM). 10-41
Air Force Cost Analysis Agency (AFCAA). 10-40
Air Force Institute of Technology (AFIT). 5-43
Air Force Operational Test and Evaluation Center (AFOTEC). 14-60
  Lessons-learned. 14-65
  Testing objectives. 14-62
  Testing tools. 14-64
Air Force Satellite Control Network Common User Element (AFSCN/ CUE). 10-65
Air Force Scientific Advisory Board. 11-29
Air Force Space and Warning Systems Center (SWSC). 10-29
Air Force Standards Improvement Executive. 2-23
Air Force STARS Demonstration Project. 15-53, 16-37
Algorithm. 8-37
Allocation. *See also Design*
  Baseline. 12-46
Amadeus Measurement System. 10-67
American Defense Preparedness Association (ADPA). 2-17
American Electronics Association (AEA). 2-17
American National Standards Institute (ANSI). 5-7, 5-59, 5-70, 13-30, 15-48
  ANSI 8652: 1995. 5-59
  ANSI/IEEE1076. 14-25
  ANSI/MIL-STD-1815A. 5-7
AN/BSY-2 Project. *See USS Seawolf (SSN 21) Submarine*
Analysis of Complexity Tool (ACT). 10-62
Anderson, Christine. 5-59, 5-61
APG-7 radar. 5-4
Application
  Engineering. 9-4
  Programming interface (API). 10-49, 13-30
  Software. 12-31
Applications Portability Profile (APP) model. 13-30
Architecture. 2-27, 3-19, 4-33, 5-27, 6-9, 10-7, 12-23, 14-28, 14-33
  Architectural Modeling Framework. 2-29
  Generic. 9-15
  Reuse of. 9-15
  Standards-based. 14-32
  Technical Architecture Framework for Information Management (TAFIM). 2-27, 2-29, 13-23, 14-32
Armstrong, C. Michael. 2-17
Army Field Manual 770-78. 4-12
Army STARS Demonstration Project. 15-53
Artificial intelligence (AI). 2-6
Assembler. 10-14
Asset Source for Software Engineering Technology (ASSET). 9-34
  Worldwide Software Resource Directory (WSRD). 9-35
Attribute. 14-41
  Process. 8-5
  Quality. 15-40, 15-45
Avionics Domain Application Generation Environment (ADAGE). 9-39

INDEX

B

B-1B Lancer. 9-13, 11-3, 15-32
  Computer Upgrade Risk Management. 6-31
  Conventional Mission Upgrade Program (CMUP). 1-28
B-2 Bomber. 2-5, 2-10, 9-13, 11-3, 12-4, 12-19
B-52 Stratofortress. 9-13
Ballistic Missile Defense (BMD). 2-12
Baseline. 11-18, 12-39, 14-21, 16-17, 16-29
  Acquisition program (APB). 12-46
  Breach. 12-46
  Budget. 15-19
  Configuration management (CM). 12-46
  Control of. 13-19
  Definition of. 12-45
  Development. 3-9
  Estimate. 12-37
  Front-loaded. 15-18
  Functional. 12-46
  Integrity. 15-18
  Product. 12-46
  Production. 3-9
  Program. 12-47
  Rubber. 15-18
  Schedule. 12-11, 13-8
  System requirements. 14-22
Baseline Practices Guide (BPG). 7-14
Battlemap Analysis Tool (BAT). 10-62
Behavior-oriented development. 14-15
Bell-Lapadula Security Policy Model. 14-74
Benchmark. 5-44, 5-66, 16-29
  Compiler. *See Compiler*
Best and final offer (BAFO). 13-45, 13-50
Best practices. 1-43, 13-36
  Software Best Practices Initiative. 2-32
  Risk Management Method. 6-24
Best-value. **13-36**, 13-40, 13-46, 13-49
Binary link. 9-17
Binary number. 2-8
Binding. 5-70, 13-26
  Dynamic. 11-12
Bischoff, Col Ron. 11-11
Boehm, Barry W. 1-23, 3-19, 6-8, 8-5, 8-41, 8-44, 8-49, 11-6, 15-58, 16-21
  COnstructive COst MOdel (COCOMO). 8-44
Boeing 777. 1-40, 2-7, 10-63
  Metrics. 8-14
Booch, Grady. 4-35, 14-38, 14-45
Borky, Col John M. (Mike). 11-11, 12-13, 12-20
Broad Agency Announcement (BAA). 12-24
Brooks, Frederick P., Jr. 1-13, 1-15, 1-20, 1-31, 16-23, 16-32
Budget. 12-11
  Baseline. 15-16, 15-19
  Budget estimate submissions (BES). 12-49
  Budgeted cost of work scheduled (BCWS). 15-16
  Contract. 15-16
  Planning, Programming, and Budgeting System (PPBS). 12-49
  President's Budget (PB). 12-49
  Program Objective Memorandum (POM). 12-49
Busey, ADM James B., IV. 2-6

C

C-130J Hercules program. 10-46, 15-63
C-141 Starlifter. 9-3
C-17 Globemaster. 2-7, 2-24, 9-3, 9-15, 12-5
C4I for Warrior. 2-25
Canadian Automated Air Traffic System (CAATS). 10-49
Capability assessment. 13-45
Capability Maturity Model (CMM℠). 7-11
  CMM℠-Based Appraisal — Internal Process Improvement. 7-5
  ISO/IEC Maturity Standard: SPICE. 7-13
  Key process area (KPA). 7-9
  People - Capability Maturity Model (P-CMM℠). 7-18
  Software Acquisition - Capability Maturity Model (SA-CMM℠). 7-21
  Software process maturity framework. 7-11
  Systems Engineering - Capability Maturity Model (SE-CMM℠). 7-26
  Systems Security Engineering - Capability Maturity Model (SSE-CMM℠). 7-23
  Trusted Software — Capability Maturity Model (TS-CMM℠). 7-26

Cargo Movement Operations System (CMOS). 14-57
Central Flow Management Unit. 10-27
Cheyenne Mountain Granite Sentry C2 upgrade. 15-62
Class. 14-42
Cleanroom engineering. 13-51, 15-49
  Acquisition. 15-56
  Correctness Theorem. 15-51
  Correctness verification. 15-51
  Defect prevention. 15-49
  Information about. 15-56
  Results. 15-53
  Usage probability distribution. 15-52
Clinton, President Bill. 2-18
Co-processing. 5-76
Cobra Dane. 10-26
Code
  Cost of. 9-16
  Reuse. *See Reuse*
  Translation. 11-16
Coding. 14-8, 14-67, 15-22
Cohen, Senator William S. 2-16
COHESION™. 4-36, 10-28
  Team/SEE. 10-28
Cohesion. 8-39
Collaboration. 14-43
Command and control (C2) software. 2-9
Command Center Processing and Display System-Replacement (CCPDS-R). 10-22, 10-24
Command, control, and communications (C3) software. 2-9
Command, control, communications, and intelligence (C3I). 2-9
Commander's Estimate of the Situation (CES). 6-36
Commercial item description (CID). 2-21
Commercial-off-the-shelf (COTS) software. 2-20, 5-29, **5-74**, 12-17, **13-21**, 13-34, 14-10, 14-22, 14-33, 14-34, 16-25
  Cautions about. 13-29
  COTS Integration and Support Model. 10-83
  Documentation. 13-31
  Integration. 5-27, 13-23
    with Ada. 13-26
  Joint Command COTS Supportability Working Group. 13-29
  Modification of. 13-32
  Support of. 11-10
Common Business Oriented Language (COBOL). 5-61
  Re-engineering of. 11-19
Common Object Request Broker Architecture (CORBA). 10-53, 13-24
  CORBA Interface Design Language (IDL). 13-25
Common Operating Environment (COE). 9-33
Communications interface (CI). 13-30
Competition. 12-16
  Design. 12-16
  Production. 12-16
Compiler. 5-11, 10-14, 10-27
  Ada. 5-38, 5-44, 5-65
  Ada 95. 5-68
  Benchmark. 5-67
    HARTSTONE. 5-68
    Performance Interface Working Group (PIWG). 5-68
  Evaluation. 5-67
    Ada Compiler Evaluation Capability (ACEC). 5-67
  Maturity. 5-44, 5-66
  Selection. 5-65
  Validation. 5-66
Completeness. 4-37, 12-5
Complexity. 1-12, 1-20, 4-34, 5-55, 8-8, 8-36, 8-38, 10-19, 11-12, 12-28
  Card Design Complexity. 8-38
  Estimation. 1-21
  Halstead Volume Metric. 8-38, 11-32
  In the RFP. 11-32
  McCabe Cyclomatic Complexity Metric. 8-38, 10-62, 11-32
  McCabe Design Complexity. 10-62
  Measurement/metrics. 8-35
Component class. 4-8
Component, software. 12-30
Comprehensive Approach for Reusable Defense Software (CARDS). 9-35
  Portable Reusable Integrated Software Modules (PRISM). 9-38
  Reuse Partnership Project. 9-37
  Training programs. 9-38
Computer Resource Technology Transition (CRTT) Program. 10-74
Computer resource utilization. 15-24
Computer Resources Control Board (CRCB). 1-40
Computer Resources Integrated Support Document (CRISD). 11-23, 11-27
Computer Resources Life Cycle Management Plan (CRLCMP). 6-35, 11-23

INDEX

Computer Resources Working Group (CRWG). 6-35, 13-4, 13-41
Computer software configuration item (CSCI). 3-4, 12-30
Computer Systems Authorization Directory (CSAD). 9-26
Computer-aided software testing (CAST). 10-55
Conahan, Frank C. 1-5
Concept Exploration and Definition Phase. 3-7, 12-13
Concept Studies Approval. 3-6
Concurrency. 3-23, 12-17, 14-41
Concurrent engineering (CE). 4-18, 14-7
Conduct of Fire Trainer (COFT). 15-53
Configuration management (CM). 2-35, 10-27, 12-46, 14-36, **15-66**
  Activities. 15-69
  Baselines. 12-46
  By contractor. 15-68
  By Government. 15-68
  Change control. 1-45
  Configuration Control Board. 15-68
  Plan. 15-68
  Requirements tracking. 15-70
  Version control. 14-20
  With Ada. 15-71
Confirmability. 4-37
Consistency. 12-5
Constraints. 12-26
  Cost. 12-14
  Schedule. 12-14, 12-40
COnstructive COst MOdel (COCOMO). 8-34, 8-41
Continuous Acquisition and Life Cycle Support (CALS). 11-24
Contract. 13-5
  Award. 13-42, **13-51**
  Budget. 15-16
  Data requirements list (CDRL). 13-13
  History. 13-40
  Protest. 13-48
  Type
    cost reimbursable. 13-8
    cost-plus. 13-34
    cost-plus award fee (CPAF). 12-20
    cost-plus incentive fee (CPIF). 12-20
    event-driven task order. 13-8
    fee-for-service. 11-24
    firm-fixed price incentive fee (FFPIF). 12-20, 13-34
    firm-fixed-price (FFP). 13-8, 13-14
    multiple award. 13-8
    schedule plus. 13-9
    selection. 13-8
    sole source. 12-16
  Work breakdown structure (WBS). 12-33
Contract Services Association. 2-17
Contractor. 13-4, 13-6
  Buy-in. 15-47
  Commitment. 1-49, 13-38
  Industry involvement. 13-10, 13-50
  Make-or-buy decision. 12-21
  Performance demonstration. 12-19
  Performance incentive. 12-20
  Prime. 13-14
  Track record. 1-48
    Domain experience. 16-22
Copying. 9-17
Cost. 8-44
  Constraints. 12-14
  COnstructive COst MOdel (COCOMO). 8-44
  Cost Analysis Requirements Document (CARD). 10-40, 12-8
  Cost/Schedule Control System Criteria (C/SCSC). 13-9, 15-15, 16-16
    actual cost of work performed (ACWP). 15-16
    budgeted cost of work performed (BCWP). 15-16
    budgeted cost of work scheduled (BCWS). 15-16
  Data. 8-21
  Driver. 10-37, 12-10, 14-69, 15-39
  Earned-value. 15-12
  Estimation. 1-21, 12-25
    Ada-specific. 8-49
    Cost estimating relationship (CER). 8-48
    independent cost estimate (ICE). 12-47
  Life cycle. 13-47, 14-69
  Of Ada. 5-39, 5-41
  Of coding. 9-16
  Of conformance. 8-49
  Of defect removal. 15-37
  Of inspections. 15-41
  Of nonconformance. 8-49
  Of quality. 9-19, 13-46, 16-7
  Of re-engineering. 11-15
  Of requirements errors. 8-5
  Of rework. 8-49
  Of support. *See Support*
  Performance. 15-16
    cost performance index (CPI). 15-21

Cost Performance Report (CPR). 8-48, 15-15
Productivity multipliers. 8-43
Proposal. 13-44
Risk. 12-42
Source selection criteria. 13-48
Costello, Robert B.. 15-3
Council on Competitiveness. 16-13
Counter analysis. 14-72
Counter measure. 14-72
Coupling. 4-36, 8-39
Creech, Gen Bill. 1-19, 1-53
Crisis Management Plan. 6-40
Critical Design Review (CDR). 3-19, 14-6, 14-28, 14-35, 14-47
Critical information. 14-71
Critical path. 12-5, 14-19
Croak, Lt Col Tom. 16-37
CrossTalk. 2-42, 4-39, 10-76, 10-99
Customer. 2-42, 9-36, 16-23
Responsibility. 6-7
Satisfaction
rating. 8-10

D

Data
Analysis. 8-20, 8-24
Data item description (DID). 13-14
Management Plan. 14-64
Objective. 8-20
Process. 8-21
Product. 8-22
Reuse. *See Reuse*
Rights. 13-34
government-purpose licence. 13-35
limited. 13-35
restricted. 13-35
unlimited. 13-34
Subjective. 8-20
Technical. 13-34
Database. 3-4, 5-72
Relational database management system (RDBMS). 14-73
Debug. 14-52, 14-55, 15-51
Debugger. 10-15
Decimal arithmetic. 10-71
Decomposition. 12-23, 12-28
Design. 12-28
Functional. 12-28, 14-15
heirarchical. 14-39
Defect. 8-49, 14-8, 14-50, 14-66
Causal analysis. 8-10, **15-33**, 15-46
Confinement. 14-30
Design insertion rate. 15-43
Detection. 8-52, 14-52
Software Defect Detection Model. 15-35
Insertion rate. 5-40
Integration. 14-52
Interface. 15-43
Mean-time-to-defect (MTTD). 5-78
Prevention. 8-8, 14-23, 15-32, 15-34
Cleanroom. 15-49
Prone modules. 15-37
Removal. 14-52, 15-41
cost. 15-37
efficiency. 8-8, 8-10, **15-37**, 15-42
rate. 14-53, 15-42
strategies. 14-55
System. 14-53
Tracking. 2-35
Unit. 14-52
Zero defects. 14-53, 16-7
Defense Acquisition Board (DAB). 1-6, 10-37
Defense Authorization Act (1991). 2-24, 13-46
Defense Information Systems Agency (DISA). 10-32, 10-71
Center for Information Management (CIM). 7-33
Defense Intelligence Agency (DIA). 14-71
Defense Management Report Decision 918. 10-71
Defense Meteorological Satellite Program (DMSP). 10-65
Defense Nuclear Agency (DNA). 4-31
Defense Plant Representative Office. 1-50
Defense Science Board (DSB). 1-11, 1-15, 13-22
Report on Acquiring Defense Software Commercially. 1-17
Report on Military Software (1987). 1-15
Defense Software Repository System (DSRS). 5-8, 9-38
Defense Standards Improvement Council (DSIC). 2-21, 12-36
Defense Systems Management College (DSMC). 12-6, 13-39
Defense Technical Information Center (DTIC). 2-28
Delivery-in-place. 11-24
DeMarco, Tom. 1-3, 8-3
Deming, W. Edward. 12-8, 15-5

INDEX

Demonstration and Validation (Dem/Val) Phase. 3-7
Demonstration Project SEE. 10-29
Department of Commerce. 16-12
Depot Maintenance Management Information System (DMMIS). 13-32
Deputy for Software Engineering (DSE). 14-65
Desert Shield. 2-25
Design. 11-10, 14-8, **14-28**, 15-22
  Architecture. 14-28, 14-31
  Competition. 12-16
  Data. 14-39
  Decomposition. 12-28
  Detailed. 13-34, 14-28, 14-38
  Diversity. 14-30
  Documentation. 14-76
    Software Design Description (SDD). 8-19
  Functional. 14-39
  Interface. 14-29
  Of data. 14-29
  Procedural. 14-29
  Reuse. *See Reuse*
  Simplicity. 14-30, 15-60
  Stability. 15-24
  To-cost. 12-18
  With Ada. 14-38
Designated Acquisition Commander (DAC). 3-16
Deutch, John. 12-6
Development. 3-9, **14-4**. *See also Life cycle: Methodology*
  Baseline estimate. 12-37
  Evolutionary. 14-9
  Incremental. 14-9
  Lessons-learned. 14-11
  Process-focused. 9-28
  Recommendations. 14-9
  Strategies
    Behavior-oriented. 14-15
    data-oriented. 14-39
    function-oriented. 14-39
    process-oriented. 14-15
    structured analysis. 14-15
Diagram
  Context. 4-45
  Decomposition. 4-45
  Entity-relationship (E-R). 4-45
Distributed Computing Environment (DCE). 13-24
Documentation. 11-32, 12-5, 14-66, **14-75**
  Ada. 14-79
  Development. 14-78
  Generator. 10-15
  Management. 14-75, 14-77
  Must-have. 14-77
  Of COTS. 13-31
  Redocumentation. 11-16
  Technical. 14-75, 14-76
DoD 4120.3-M. 2-21
DoD 5000.2-R. 2-24
DoD Index of Specifications and Standards (DoDISS). 2-22
DoD SD-2. 2-21
DoD-STD-1703. 2-23, 15-48
DoD-STD-2167A. 2-23, 5-27, 6-32
DoD-STD-7935A. 2-23
DoDD 3405.1. 2-24, 5-79
DoDD 5200.28. 14-74
DoDI 5000.2. 3-13, 4-27
Domain. 4-5
  Analysis. 4-7
  Design. 4-8
  Dictionary. 4-8
  Domain-Specific Software Architecture (DSSA)
    Avionics Domain Application Generation Environment (AD-AGE). 9-39
  Experience. 16-22
  Horizontal. 4-6
  Identification. 4-7
  Implementation. 4-9
  Model. 9-15
  Problem. 9-15, 14-45
  Solution. 14-47
  Vertical. 4-6
Domain engineering. 4-5, 9-4
  Applications engineering. 9-4
  Benefits of. 4-11
  Model. 9-15
Domain-Specific Software Architecture (DSSA). 9-39

E

Earned-value. 15-12
Editor. 10-15
Edmonds, Maj Gen Albert J. 2-26, 5-8, 5-11, 5-82, 16-8
EF-111A Electronic Fox
  System Improvement Program. 10-46
Effectiveness. 14-63
Efficiency. 4-32
  Input/output. 4-33

Memory. 4-32, 12-48
Run-time efficiency (RTE). 4-33
Source code. 4-32
Effort. 15-23
Manpower buildup index (MBI). 16-24
Electronic Data Interchange (EDI). 15-18
Electronic Library Services and Applications (ELSA). 9-39
Electronics Industries Association (EIA). 2-17
EIA Standard 632. 4-13
Embedded Computer Resources Support Improvement Program (ESIP). 10-74
Embedded software. 2-10
Encapsulation. 14-34, 14-41
Engineering and Manufacturing Development (EMD) Phase. 3-9
Engineering change proposal (ECP). 3-23, 14-6
Ericsson AB Telecommunications Operating System. 15-54
Error. 14-30, 14-52, 16-7
Cost of. 1-26, 8-5
Prevention. 14-51, 15-28
Recovery. 14-30
Estimation. 1-22, 10-7
Accuracy. 12-40
Baselines. 12-37
Complexity. *See Complexity*
Cost. *See Cost*
Estimate at completion (EAC). 13-9, 15-20
Inadequate. 1-20
Independent cost estimate (ICE). 12-47
Parameters. 12-41
Selection of. 12-42
Single point. 12-40
Size. *See Size*
Evolutionary
Development. 3-13, 12-10, 14-6, 14-9, 14-15
Requirements. 14-20
Evolutionary Spiral Process (ESP). 15-62
Excellence
Standard-of. 16-7
Exception. 14-42
Exception handling. 4-32

F

F-111 Aardvark. 2-10
F-117 Stealth Fighter. 2-5, 9-13
F-14 Tom Cat. 6-29, 11-23
F-15 Strike Eagle. 5-3, 9-13, 11-23
VHSIC Central Computer (VCC). 6-34
F-16 Fighting Falcon. 2-5, 2-10, 9-3, 9-13, 11-23, 14-60, 15-12, 15-35
Modular Mission Computer. 9-24
Reuse. *See Reuse*
Upgrade Program. 9-24
F-22 Advanced Tactical Fighter. 1-6, 1-35, 1-56, 2-6, 4-19, 5-8, 8-40, 9-3, 9-13, 10-46, 11-11, 11-21, 12-4, 12-35, 13-17, 14-16, 15-62, 15-69
Software engineering environment (SEE). 10-28
Work breakdown structure (WBS). 12-35
F-4 Falcon. 2-5
F/A-18 Hornet. 2-4, 11-23
Fagan Inspection Process. 15-45
Failure. 14-66
Fain, Lt Gen Jim. 2-6
Fault. 14-66
Rate. 5-41
Tolerance. 4-31, 14-30
Feature point. 8-33, 8-37
Federal Acquisition Computer Network. 2-19
Federal Acquisition Regulations (FAR). 2-27, 13-8
FAR Sup 52-227-14. 13-34
Federal Acquisition Streamlining Act of 1994. 2-18
Federal Aviation Administration (FAA). 5-38, 12-13
Advanced Automation System (AAS). 9-25
Federal Information Processing Standards (FIPS). 5-70, 14-32
FIPS 119-1. 5-59
Fee-for-service. 11-24
Field Artillery Tactical Data System (FATDS). 5-14
Firm-fixed-price incentive fee (FFPIF) contract. 13-34
Flight Plan for Success. 15-9
Fogleman, Gen Ronald R. 1-37, 1-55
Fornell, Lt Gen Gordon E. 2-26
Forward engineering. 11-16
Full operational capability (FOC). 3-20, 12-46

INDEX

Function point. 8-33, 8-35
International Function Point Users Group (IFPUG). 8-35
Funding. 12-11

G

General Accounting Office (GAO). 1-7, 2-15
Reports. 1-7
General Services Board of Contract Appeals (GSBCA). 13-48
Generic Package of Elementary Functions (GPEF). 5-71
Generic Package of Primitive Functions (GPPF). 5-71
Generics. 11-12
Gilligan, John. 8-29
Global Combat Support System (GCSS). 2-14, 9-33
Global Command and Control System (GCCS). 9-33
Global Positioning System (GPS). 10-65
Goldplating. 12-22, 15-60
Government-furnished-software (GFS). 9-42
Government-off-the-shelf (GOTS). 14-33
Grand design (waterfall). 3-21, 10-26
Graphical Kernel Systems (GKS). 5-70
Graphics Standard Interface Standard (GSIS). 5-73
Greene, Col Joseph, Jr. 1-6
Guenther, LGEN Otto. 2-14
Gulf War. *See Operation Desert Storm*

H

Halstead Volume Metric. 8-38, 11-32
Hardware. 4-3
Requirements. *See Requirements, hardware*
Resources. 8-45
Selection. 14-26
Harris Corporation. 6-18
HARTSTONE benchmark. 5-68
Hekman, RADM John G. 2-25
Hierarchy. 14-43
House Armed Services Committee. 1-35
Hughes Aircraft Company. 7-42
Human computer interface (HCI). 13-30
Style Guide. 2-30
Humphrey, Watts. 3-9, 3-14, 7-4

I

IBM AOEXPERT/MVS. 15-55
IBM COBOL Structuring Facility. 15-54
IBM Tape Drive Microcode Project. 15-55
Implementation. 14-8, 14-67
Incentive, contractual
Cost-plus award fee (CPAF). 12-20
Cost-plus incentive fee (CPIF). 12-20
Firm-fixed price incentive fee (FFPIF). 12-20, 13-34
Incremental development. 3-16
Independent cost estimate (ICE). 12-47
Independent verification and validation (IV&V). 15-28, 15-48
Indicators
Management. 8-4
Reliability. 8-4
Information Age. 2-3, 4-3
Information engineering (IE). 4-40
Architecture. 4-42
Process. 4-42
Information hiding. 4-35, 9-18, 14-42
Information Resource Dictionary System (IRDS). 5-69
Information storage and retrieval interface (ISRI). 13-30
Information Technology Management Reform Act of 1994. 13-48
Information Technology Standards Guidance (ITSG). *See Standards*
INFORMIX. 14-74
Online Secure Product. 14-75
Inheritance. 11-12, 14-43
Initial operational capability (IOC). 3-20, 12-46, 13-8
Inspection, peer. 2-36, 15-38
Author. 15-46
Benefits of. 15-40
Buy-in. 15-47
Cost of. 15-41
Efficiency. 15-37
Exit criteria. 15-44
Fagan Inspection Process. 15-45
Metrics. 8-10
Moderator. 15-46
Objectives. 15-41
Process. 15-44
Reader. 15-46
Tester. 15-46
Training. 15-48

Installation Materiel Condition Status Reporting System (IMCSRS). 5-8
Institute for Defense Analyses (IDA). 5-19, 15-41
Institute of Electrical and Electronics Engineers (IEEE). 5-7, 5-45, 8-28, 11-3, 13-30, 15-48
IEEE 1220. 4-14
IEEE Std.610.12.1990. 5-45
Instruction set. 2-8
Instructions to Offerors (ITO). 11-30
Intangibility. 1-28
Integrated Computer-Aided Manufacturing Definition Language (IDEF). 4-44, 14-16
Integrated Computer-Aided Software Engineering (I-CASE). 10-32
Integrated Logistics Support (ILS). 11-28
Plan (ILSP). 12-13
Integrated Master Plan (IMP). 3-7, 12-36
Integrated Master Schedule (IMS). 3-7, 12-36
Integrated product development (IPD). 1-40, **4-16**, 4-28, 11-21
Integrated Weapon System Management (IWSM). 11-21
Integrated Weapons System Support Facility (IWSSF). 11-23
Integration. 14-19, 14-67
Defect rate. 5-40
Defects. *See Defect*
Error rate. 1-26
Of COTS. 13-23
Rules for. 14-34
Intelligence software. 2-11
Intelsat I-VII Satellite Project. 5-34
Interface. 2-34, 5-61, 14-19, 14-32, 14-42, 16-21
Application program interface (API). 13-30
Between Ada components. 5-9
Communications interface (CI). 13-30
Human computer interface (HCI). 13-30
Information storage and retrieval interface (ISRI). 13-30
Interface Control Document (IRD). 14-19
Interface Definition Language (IDL). 10-54
Interface Requirements Specification (IRS). 14-19
Requirements. 14-19
Interference. 5-11
Internal Revenue Service (IRS). 13-49
International Function Point Users Group (IFPUG). 8-35
International Standards Organization (ISO). 5-7, 5-70, 13-30
ISO/IEC 12207. 3-5
ISO/IEC Maturity Standard: SPICE. 7-13
ISO/OEC 8652: 1995 (E). 5-59
Interoperability. 11-6, 14-19, 14-32, 14-34
Risk. 13-30

J

Jabour, Lt Col W. Jay. 12-20
Japanese software development. 10-3, 15-8
House-of-quality. 16-5
Job order. 12-33
Joint Computer-aided Acquisition and Logistic Support (J-CALS). 13-7
Joint venture partnership. 13-17
Jones, Capers. 1-30, 1-32
Jules Own Version of the International Algebraic Language (JOVIAL). *See Language, programming*
Just-in-Time training program. 15-65

K

Kaminski, Paul. 2-18
Key process area (KPA). 5-79, 7-7

L

Lampe, Col George P.. 15-63
Language, programming. 11-32
Ada. *See Ada*
Assembly. 14-45
Fourth generation language (4GL). 5-45
Higher order language (HOL). 5-57, 14-22
Common Business Oriented Language (COBOL). 5-8, 5-61, 11-18
Formula Translation (Fortran) Language. 5-61, 9-21, 10-25
Jules Own Version of the International Algebraic Language (JOVIAL). 11-19
Interface definition language (IDL). 10-54

INDEX

Program design language (PDL). 14-38
Standardization. 5-7, 5-40, 14-4, 14-32
Third generation language (3GL). 5-45
Leadership. 1-53
Leong-Hong, Belkis. 5-9
Librarian. 10-15
Life cycle. **3-1**, 11-3
Methodology. 3-11
choosing among. 3-21
Phases. 3-6
Software
cost. 14-69
Software Support (LCSS). 11-21
System. 3-4
Line-of-code (LOC). 8-34
Linkage. 10-17
Linker. 10-14
Litton Data Systems. 7-44
Loader/reformatter. 10-15
Localization. 4-36, 14-42
Logistics
Air Force Logistics Information File (AFLIF). 14-57
Integrated Logistics Plan (ILP). 12-13
Support analysis (LSA). 11-19, 13-31. *See also Life cycle*
Software Supportability Checklist. 11-19
Longuemare, Noel. 2-24, 2-32
Loral Federal Systems. 8-9
Low Altitude Navigating Infrared for Night (LANTIRN). 5-4
Ludwig, Lt Gen Robert H. 2-4, 5-3
Lyons, Col Robert, Jr. 4-19, 8-40, 10-28, 16-17

M

M1A1 Abrams. 2-4
Maintenance. 11-5, 14-8. *See Support*
Maintainability. 8-28
Major Automated Information System Review Council (MAISRC). 3-16, 6-31, 10-37
Make-or-buy decision. 12-21
Malcolm Baldridge Award. 1-43
Management. 1-17, 1-45, 1-53, 4-37, 6-6, 15-5
Challenge. 16-32
Commitment. 13-38
Metrics. *See Measurement/metrics*
Of design. 14-6
Of PDSS. 11-25, 16-15
Of requirements. 14-20
On-going program. 16-9
Oversight. 1-45
Program Management Support System (PMSS). 10-42
Proposal. 13-43
Tools. *See Tools*
Troubled program. 16-16
Management information system (MIS). 5-60, 11-17
Ada MIS. 5-6
Market analysis. 12-23, 12-36
Martin, James. 4-41, 5-46
MathPack. 10-60
Maturity. 1-50, **7-3**
Ada. 5-40, 8-43
Benefits. 7-29
Growth curve. 12-47
In the RFP. 7-46
Models. 7-9
Of software. 14-64
severity point system. 14-64
McCabe
Cyclomatic Complexity Metric. 8-38, 10-62, 11-32
Design Complexity. 10-62
tool. 10-60
Instrumentation Tool. 10-62
Slice Tool. 10-62
McPeak, Gen Merrill A.. 12-4
Mean-time-to-defect (MTTD). 5-78
Mean-time-to-failure (MTTF). 15-49
Measurement/metrics. 8-3, 8-11, 11-26, 12-25, 15-5, 16-17, 16-28
Boeing 777 Metrics. 8-14
Cautions about. 8-51
Data analysis. 8-20
Data collection. 8-18
Earned-value. 15-21
In the RFP. 8-53
Life cycle. 8-7
Loral Federal Systems. 8-9
Management. 8-5, 15-8
Metrics Usage Plan. 8-53
OPSEC measures. 14-72
Peer inspection metrics. 8-10
efficiency. 8-10
Process. 8-5, 8-11, 15-8
Product. 8-5
Quality. 8-10, 14-77, 15-8, 15-25, 15-32, 16-28
Selection. 8-17
Tools. *See Tools*
Megaprogramming. 4-51

Message. 14-43
MIL-STD-1388A. 11-23
MIL-STD-1750A. 5-34
MIL-STD-1815A. 6-32
MIL-STD-498. 2-23, 2-47, 3-5, 9-43, 13-43, 15-28
MIL-STD-961C. 2-21
Milestone. 3-12, 12-45
  Completion. 12-20
  Decision. 3-3, 14-20
    Milestone Decision Authority (MDA). 2-21, 3-6, 12-36, 12-47
  Performance. 8-46
Military specification (MilSpec). 2-25, 13-36
Military standard (MilStd). 2-25, 13-36
Mission Area Assessment (MAA). 3-6
Mission Area Plan (MAP). 3-6
Mission definition. 12-14
Mission Need Analysis (MNA). 3-6
Mission Need Statement (MNS). 3-6
Mobile Space Project. 9-32
Models. 4-28, 8-21, 10-37
  Ada. 5-78
    Spiral Model Environment. 3-20
  Ada Process Model. 15-53
  Air Force Acquisition Model (AFAM). 10-41
  Applications Portability Profile (APP) Model. 13-30
  Bell-Lapadula Security Policy Model. 14-74
  COnstructive COst MOdel (COCOMO). 8-34, 8-41
  COTS Integration and Support Model. 10-83
  DoD Enterprise Model. 4-25
  Domain. 9-15
  Feature or functional. 4-8
  Information. 4-8
  Integrated Computer-aided Manufacturing Definition Language (IDEF). 4-44
  Maturity. *See Maturity*
  Operational. 4-8
  Parametric. 8-47, 10-37
    selection. 10-37
  Reuse Strategy Model: Planning Aid for Reuse-based Projects. 9-29
  Software Defect Detection Model. 15-35
  Spiral Model. 3-19
Modifiability. 4-30, 11-6
  Of COTS. 13-32
Modified condition decision coverage (MCDC). 10-63
Modularity. 4-35, 14-42
Module. 4-35
Mortar Ballistics Computer (MBC). 15-53
Mosemann, Lloyd K., II. 1-4, 1-44, 4-20, 4-26, 4-46, 5-1, 10-38, 13-22, 13-36, 16-3, 16-33
Motif. 5-70, 5-73

N

National Aeronautical Space Agency (NASA)
  Goddard Space Flight Center. 15-55
  Satellite Control Projects. 15-55
National Computer Security Classification (NCSC). 14-73
National Institute of Standards (NIST). 13-30, 14-32, 15-48
National Science Foundation (NSF). 1-11
National Security Agency (NSA). 7-26
National Security Industrial Association (NSIA). 2-17
National Software Data and Information Repository (NSDIR). 8-22
  History. 8-25
Naval Aviation Logistics Command Management Information System (NALCOMIS). 6-30
Naval Coastal Systems Station. 15-53
Naval Command and Control Ocean Surveillance Center (NCCOSC). 5-14
Navy Computer and Telecommunications Area Master Station (NCTAMS). 5-30, 5-33
Navy Management Systems Support Office (NAVMASSO). 6-31
Navy Standards Improvement Executive. 2-23
Navy Tactical Data System (NTDS). 5-13
New-Start Program. 16-9
Non-developmental item (NDI). 14-33
Nuclear Mission Planning and Production System (NMPPS). 1-25, 14-7
Numerics. 5-61

INDEX

O

O'Berry, Lt Gen Carl G. 2-15, 10-25
Object. 5-57, 14-40
Object Management Group (OMG). 10-54
Object Request Brokers (ORB). 13-24
Object-oriented. 14-40
  Design (OOD). 11-12, 14-15, 14-40
    Common Object Request Broker Architecture (CORBA). 10-53
  Development (OOD). 4-35, 5-60, 10-52, 14-15, 15-60
  Reuse. 14-40
Objectives. 12-13, 12-24, 12-26, 12-46
  Planning. 12-26
Office of Management and Budget (OMB)
  Circular A-109. 3-3
Oklahoma City Air Logistics Center (OC-ALC). 7-34
Open systems. 2-26
  Architecture. 2-27, 14-27, 14-33
  Definition of. 2-26
  Open systems environment (OSE). 2-26, 5-70, 10-24
  Interoperability. 14-32, 14-34
OpenLook. 5-70, 5-73
Operating system. 5-72, 14-27
Operation Desert Storm. 1-56, 2-1, 2-3, 2-25, 5-30, 5-82, 14-73, 15-15
Operational Readiness Document (ORD). 12-46
Operational Requirements Document (ORD). 3-6, 3-7, 14-21, 14-71
  Operational Requirements Working Group (ORWG). 14-21
Operational test and evaluation (OT&E). 14-62
Operations and Support Phase. 3-10
Orange Book. 14-74
Oregon Graduate Institute Formal Methods Research. 10-78
Owens, ADM William A. 2-15

P

Package specification. 5-70
Paige, Emmett, Jr. 2-24, 2-27, 2-32, 5-8, 5-58, 5-61, 16-33
Paradigm. 4-28
Parnas, David. 1-11, 1-48
Partitioning. 14-33, 14-39
Patriot missile. 2-4
People. 1-43, 1-47, 12-5, 13-38. *See also Team*
  Communications among. 1-26
  Key. 13-37, 14-7
  Letter of Commitment. 13-37
  Skills/talent. 8-43, 9-16, 14-31
    Skills Matrix. 9-16, 13-38
  Staffing constraints. 12-40, 16-22
  Turn over. 10-8
People - Capability Maturity Model (P-CMM[SM]). 7-18
  Structure. 7-19
Performance
  Goals. 12-10
  Incentive. 12-20
  Interface Working Group (PIWG) benchmark. 5-68
  Specification. 2-21
  Standard. 16-8
  Standard performance specification. 2-21
  Threshold. 12-47
Perry, SECDEF William J. Jr. 2-14, 2-17, 2-20, 2-22, 6-32, 12-36, 13-21
  1994 Memo. 2-25, 2-32, 13-36
Personnel. *See People*
Pilot Vehicle Interface (PVI). 11-22, 15-15
Planning. 10-7, 12-3, 12-23
  Constraint. 12-26
  Contingency. 12-10
  For security. 14-68
  For source selection. 13-41
  For support. 11-11
  Objectives. 12-24, 12-26
  Process. 12-23
  Programming, and Budgeting System (PPBS). 12-49
  Scope. 12-24, 12-26
  Strategic. 12-4
    Plan. 12-42
Pointer. 5-60
Polymorphism. 14-43
Portable Reusable Integrated Software Modules (PRISM). 9-38
  Command Center Store. 9-38
Position Location Reporting System. 15-53
Post-deployment software support (PDSS). *See Support*
Powell, GEN Colin L. 1-34, 2-3, 16-3
Pre-planned product improvement (P3I). 12-10, 12-22, 12-47

Preliminary Design Review (PDR). 3-19, 14-6, 14-28, 14-36
President's Budget (PB). 12-49
Preston, Colleen. 2-16
Prime mission product. 12-30
  Summary WBS. 12-31
Primitive. 5-60
Procedures. 3-4
Process. 2-22, 7-3, 11-26, 12-6, 13-6, 15-9, 16-17
  Ada Process Model. 15-53
  Attribute. 8-5
  Configuration Management Software (PCMS). 10-70
  Definition. 7-3
  Effectiveness. 15-8
  Entry and exit criteria. 8-18
  Evaluation. 15-12
  Focused approach. 9-28, 12-6, 14-7, 15-6
  Improvement. 15-1, 16-6, 16-26
    continuous. 15-5, 15-42
    Process Improvement Plan. 15-4, 15-5
  Metrics. 8-5
  Normalization. 13-49
  Oriented-development. 14-15
Process Action Team
  Report on Military Specifications and Standards (1994). 1-11, 2-20
  Software Process Action Team Report (1992). 1-16
Processor
  Host. 5-72
  Target. 5-72
ProcessWeaver®. 10-46, 10-79, 13-13, 13-42
Product
  Attribute. 8-5
  Data. 8-22
  Metrics. 8-5
Product-line. 9-9, 9-32
  Benefits. 9-10
  Paradigm shift. 9-11
Production and Deployment Phase. 3-10
  Competition. 12-16
Productivity. 7-31, 8-42, 13-38, **15-57**
  Ada. 5-39, 5-41
  During support. 11-5
  Increasing. 15-41, 15-43
  Index (PI). 16-24
  Rate. 8-49
  Software Productivity Consortium (SPC). 15-62
Professional Services Council. 2-17
Program
  Catastrophe. 16-27
  New-start. 15-55, 16-9
  Objective. 12-24
    Program Objective Memorandum (POM). 12-49
  On-going. 15-56, 16-9
  PDSS. 16-15
  Program Trouble Report (PTR). 8-10
  Scope. 12-24
  Troubled. 15-56, 16-16
Program Executive Officer (PEO). *3-16*
Program Office Estimate (POE). 6-33
Programmed Depot Maintenance Scheduling System (PDMSS). 10-44
Programmer
  American. 10-3
  Indian. 10-3
Programmers Hierarchical Interactive Graphics System (PHIGS). 5-70
Programming, multi-version. 4-31
Protection. 14-73
Protest. 13-48
Prototyping. 14-15, 14-21, 15-61
  Benefits. 14-23
  Cautions about. 14-23
  Increasing productivity with. 15-61
  Pre-contract award. 14-23
Public Ada Library (PAL). 10-72
Putnam, Lawrence H.. 16-24

Q

Quality. 1-44, 7-30, 8-10, 8-27, 12-20, 14-3, 15-3, 15-7
  Attributes. 8-28, 14-18, 15-40, 15-45
  Control. 15-28
  Cost of. 9-19, 13-46, 16-7
  Design. 14-28
  Estimation of. 5-78
  Objectives. 14-16
  Software quality assurance (SQA). 1-45, 14-51, 14-55, 15-28, 16-31
  Software Quality Framework. 16-23
  Standard-of. 16-28
  Tracking. 15-70
  Vision-for. 16-26
Quann, Eileen Steets. 4-21, 5-15

INDEX

R

Raggio, Maj Gen Robert. 1-37
Rapid Open Architecture Distribution System (ROADS). 11-38
Rate monotonic analysis (RMA). 10-55
  Generalized Rate Monotonic Scheduling (GRMS). 10-57
  Rate Monotonic Analysis for Real-Time Systems (RMARTS). 10-57
  Scheduling (RMS). 5-29
Rational. 10-24, 14-36
  Apex™. 10-27, 10-49, 15-53
  Environment™. 5-75, 5-78, 10-24, 10-27, 10-49, 13-42, 15-71, 16-23
Raytheon. 7-40
Re-development. *See Support*
Re-engineering. 4-10, **11-13**
  Cost. 11-15
  Of COBOL. 11-19
  Of JOVIAL. 11-19
  Process. 11-15
  Re-engineering Tools Report. 11-19
  Software Re-engineering Center (SRC). 11-19
  To Ada. 11-17
Reagan, President Ronald. 6-31
Real-time. 5-60
  Processing. 2-10
RECORDER. 11-18
Recovery block. 4-31
Redundancy. 14-19
Reifer, Don. 5-12
Relational database management system (RDBMS). 5-72
Reliability. 2-10, **4-30**, 5-78, 8-28, 12-20, 14-30
  Indicator. 8-4
Remote Procedure Call (RPC). 13-24
Report of the Industry Panel on Specifications and Standards. 2-20
Repository, deliverables. 15-12
Repository-Based Software Engineering (RBSE). 9-39
Representation. 14-18
Request for Information (RFI). 12-24
Request for Proposal (RFP). 14-35
  Addressing in the RFP
    Ada. 5-78
    maturity. 7-46
    measurement. 8-53
    reuse. 9-42
    risk. 6-43
    software support. 11-29
    tools. 10-79
  Contract data requirements list (CDRL). 13-13
  Data item description (DID). 13-14
  Development of. 13-10
  Evaluation of. 13-45
    best and final offer (BAFO). 13-45, 13-50
    best-value. **13-36**, 13-40, 13-46
    cost proposal. 13-44
    management proposal. 13-43
    technical proposal. 13-42
  Software considerations. 13-18
  Software supportability. 11-29
  Statement of Work (SOW). **13-13**
Requirements Engineering Environment (REE). 10-47
Requirements, hardware. 14-25
  Quantitative performance requirement (QPR). 14-27
Requirements, software. 1-17, 10-7, 14-8, **14-15**, 15-22
  Ada. 5-77
  Change process. 14-20
  Component. 14-56
  Cost Analysis Requirements Document (CARD). 10-40, 12-8
  Cost of errors. 8-5
  Creep. 8-40
  Down-scoping. 1-22, 12-18
  Evolutionary. 14-20
  Explicit. 8-41
  Freezing of. 14-20
  Implicit. 8-41, 14-33
  Incremental build-up of. 14-20
  Management of. 14-20
  Minimally acceptable. 14-6
  Overloading. 14-43
  Stability. 1-23, 1-45, 14-20, 15-24
  Traceability/tracking. 14-20
  Validation. 14-16, 14-21
Requirements, system. 12-47
  Baseline. 14-22
  Operational. 14-6
Resources
  Estimation. 12-24
  Hardware. 8-45
  Manpower. 8-45, 15-44
    manpower buildup index (MBI). 16-24
  Reusable. 8-46
  Software. 8-46
  Tracking. 15-12

Restructuring. 11-16, 11-18
Retarget. 11-16
Reusable Integrated Command Center (RICC). 9-41, 15-53
Reuse. 5-74, 9-3, 9-17, **15-60**. *See also Product-line*
  Ada. 9-18
  Air Force Software Reuse Implementation Plan (1992). 9-14
  And object-oriented development. 14-40
  Assets. 8-46
  Cost of. 9-19
  Cost/benefits. 9-18
  Design. 9-16
  DoD Reuse Initiative Program. 9-43
  DoD Software Reuse Initiative Program. 9-7
  Embedded weapon systems. 9-12
  Engineering for. 9-4
  F-16 reuse. 9-24
  F-22 reuse. 9-23
  Hewlett-Packard (HP). 9-24
  Implementation of. 9-7
  Implementation Plan. 9-8
  In the RFP. 9-42
  Of architectures. 9-15
  Of code. 9-16
  Of data. 9-18
  Of designs. 9-16
  Of specifications. 9-15
  Opportunities. 9-12
  Process. 9-4
  Programs. 9-26
    Asset Source for Software Engineering Technology (ASSET). 9-34
    Comprehensive Approach to Reusable Defense Software (CARDS). 9-35
    Electronic Library Services and Applications (ELSA). 9-39
    Repository-Based Software Engineering (RBSE). 9-39
    Reuse Partnership Project. 9-37
    Software Technology for Adaptable, Reliable Systems (STARS). 9-27
  Prototyping with. 14-22
  Repository. 9-7, 9-34
    Asset Source for Software Engineering Technology (ASSET). 9-34
    Defense Software Repository System (DSRS). 5-8, 9-38
    Interoperability among. 9-39
    Portable Reusable Integrated Software Modules (PRISM). 9-38
    Reusable Ada Products for Information Systems Development (RAPID). 5-8, 10-71
  Reuse Project Officer (RPO). 9-8
  Reuse Strategy Model: Planning Aid for Reuse-based Projects. 9-29
  Standard Systems Center (SSC). 9-22
Reverse engineering. 11-15
Review. 15-38
  Informal. 15-12
Rework. 1-20, 8-5, 8-49
  Cost of. 8-49, 15-33
Rice, Secretary Donald B. 1-35, 12-4
Risk
  Assumption. 6-10
  Avoidance. 6-10
  B-1B Computer Upgrade Risk Management. 6-31
  Best Practices Risk Management Method. 6-24
  Boehm's Risk Management Method. 6-19
  Champion. 6-18
  Contingency plan. 6-38
  Control. 6-10
  Crisis Management Plan. 6-40
  Element tracking. 6-41
  Factors. 6-6
    environmental. 6-8
    interrelated. 6-9
  Integrated Risk Management Process (IRMP). 6-34
  In the RFP. 6-43
  Management. 2-34, **6-4**, 12-24, 13-19
    methods. 6-12
    paradigm. 6-20
    process. 6-11
  Mitigation. 6-10
  Of software. 6-5
  Performance Risk Analysis Group (PRAG). 6-46
  Risk Aversion Plan (RAP). 6-36
  Risk Estimate of the Situation (RES). 6-36
  Risk Management Plan (RMP). 5-66, 6-35, 14-24
  Security. 14-72
  Software Risk Evaluation (SRE). 6-13
    Software Development Risk Taxonomy (SDRT). 6-13
    Taxonomy-Based Questionnaire (TBQ). 6-14
  Team Risk Management. 6-27
  Top-10 List. 6-8, 6-21, 16-20

INDEX

Transference. 6-11
Roetzheim, William H. 6-38
Rome Air Development Center (RADC). 8-41
Rome Laboratory. 1-26, 10-77
Software Quality Framework. 16-23
Royce, Walker. 10-26
Run-time
Environment (RTE). 5-72, 5-77
System (RTS). 10-14

S

Safety. 4-31, 5-61, 14-68
Multi-version programming. 4-31
Safeguards. 4-31
Salvucci, Anthony. 1-57, 13-13
Sammi Development Kit (SDK). 10-49
Schedule. 6-7, 8-44, 12-47, 13-9, 16-19, 16-23
Baseline. 12-11, 12-45, 13-8
Budgeted cost of work scheduled (BCWS). 15-16
Changes. 1-22
Compression. 16-20
Constraints. 12-14, 12-40
Development. 12-11
Drivers. 10-37, 12-10
Estimation. 1-21, 12-25
Integrated Master Schedule. 12-36
Metrics. 8-46
Milestone performance. 8-46
Performance. 15-16
milestone performance. 8-46
on-time completion. 12-9, 12-10
schedule performance index (SPI). 15-20
Risk. 12-42
Schedule-plus contract. 13-9
Slips. 1-22
Tracking. 2-35
Schwarzkopf, GEN H. Norman. 12-3, 16-10
Scientific American. 1-10, 4-21, 10-3, 10-8, 16-11
Scope. 12-24, 12-26
Statement of. 12-26
Sears, RADM Scott L. 5-22
Security. 5-61, 13-25, 14-68
Bell-Lapadula Security Policy Model. 14-74
Critical information. 14-70
National Computer Security Classification (NCSC). 14-73
Operations security (OPSEC). 14-70
OPSEC measures. 14-72
OPSEC plan. 14-70
OPSEC process. 14-70
Orange Book. 14-74
Risk assessment. 14-72
System Security Officer (SSO). 14-75
System Threat Assessment Report (STAR). 14-71
Threat. 14-71
Trusted Computer System Evaluation Criteria (TCSEC). 14-74
Vulnerability. 14-71
Semantics. 5-7
Sergeant York. 12-17
Severity point system. 14-64
Shalikashvilli, GEN John. 13-21
Shewhart Cycle. 15-6
Short-range Attack Missile II (SRAMII). 15-32
Side-effect. 11-8, 13-30
Silver Bullet. 1-13, 1-30, 5-50, 6-7
Simulation, hardware. 14-25
Single-Stage Rocket Technology (SSRT). 12-17
Size. 6-9, 15-23
Estimation. 1-21, 12-24, 16-21
In the RFP. 11-32
Measurement. 8-32
feature point. 8-37
function point. 8-35
lines-of-code (LOC). 8-34
Reduction. 16-25
Skills Matrix. 9-16, 13-38
SNAPSHOT. 11-18
Society of Automotive Engineers (SAE). 11-28
Software Capability Evaluation (SCE). 5-79, 7-7
Key process area (KPA). 5-79, 7-7
Software crisis. 11-5
Software Design Document (SDD). 15-22
Software development. *See Development*
Software Development Capability Evaluation (SDCE). 7-7
Software Development Capability/ Capacity Review (SDCCR). 1-37
Software Development Plan (SDP). 1-30, 1-36, 1-51, 2-23, 3-12, 5-27, 6-36, 8-5, 11-30, 12-11, 12-24, **14-8**, 15-12, 16-16
Software engineering. 1-49, **4-3**, 4-20, 4-26, **4-47**, 5-9, 13-3, 16-3

Discipline. 4-4, 4-28, 11-9
Forward engineering. 11-16
Goals. 4-29
Information about. 4-39
Management of. **4-37**
Methods. 4-27
Principles. 4-33
and Ada. **5-9**
Procedures. 4-28
Restructuring. 11-16
Reverse engineering. 11-15
Software engineering environment (SEE). *See Tools*
Software Engineering Institute (SEI). 4-20, 5-5, 5-39, 7-7, 10-76
Software Engineering Laboratory (SEL). 9-21, 15-55
Software Engineering Process Group (SEPG). 6-18
Software Process Assessment (SPA). 7-5
Software Process Improvement Capability dEtermination (SPICE). 7-13
Baseline Practices Guide (BPG). 7-14
Capability levels. 7-15
common feature. 7-16
generic practice. 7-16
Product suite. 7-14
Software Product Evaluation (SPE). 11-24
Software Productivity Consortium (SPC). 15-62
Consortium Requirements Engineering. 15-63
Software Program Managers Network (SPMN). 2-36, 6-6, 15-9, 16-18
Flight Plan for Success. 15-9
Software project summary WBS. 12-31
Software Re-engineering Center (SRC). 11-19
Software Requirements Specification (SRS). 1-29, 13-14, 14-18, 14-22, 14-31, 14-33, 15-22
Software Support Activity (SSA). 13-41
Software Technology for Adaptable, Reliable Systems (STARS). 4-11, 9-27, 15-56
Air Force STARS Demo Project. 15-53
Army STARS Demo Project. 15-53
Space Command and Control Architectural Infrastructure (SCAI). 9-29
Software Technology Support Center (STSC). 2-41, 4-39, 5-75
Re-engineering support. 11-19
Re-engineering Tools Report. 11-19
Services. 2-41
Source selection. 13-39
Best-value. 13-46, 13-49
Contract award. 13-42, 13-51
Criteria. 13-12, 13-36
corporate experience. 13-40
cost. 13-48
Letter of Commitment. 13-37
technical. 13-48
Evaluation Board (SSEB). 13-40, 13-51
Planning. 13-41
Pre-Validation Phase. 13-41
Sources Sought Announcement. 12-24
Space and Warning Systems Center (SWSC). 4-51
Space Command and Control Architectural Infrastructure (SCAI). 4-11, 4-51, 9-29
Space Shuttle. 8-12, 14-5, 14-46, 14-53, 15-39, 15-67
Specification
Guide. 2-21
Performance. 2-21
Program-unique. 2-21
Reuse of. 9-15
Standard performance. 2-21
Specifications & Standards — A New Way of Doing Business. 2-20
Spiral development. 3-19
Ada Spiral Model Environment. 3-20
SQL Ada Module Description Language (SAMeDL). 5-72
SQL Ada Module Extension (SAME). 5-72
Stability. 12-8
Standard Army Management Information Systems (STAMIS). 5-8
Standard Systems Center (SSC). 9-22, 10-76, 14-57
Reuse. *See Reuse*
Standards. 1-52
1553 databus. 5-69, 10-71
Ada. 5-11
Ada Semantic Interface Specification (ASIS). 5-71, 10-73
Adopted Information Technology Standards (AITS). 2-29
Generic Package of Elementary Functions (GPEF). 5-71
Generic Package of Primitive Functions (GPPF). 5-71

INDEX

Government Open Systems Interconnection Profile (GOSIP). 5-70, 10-71, 14-32
Graphical Kernel Systems (GKS). 5-70
Graphics Standard Interface Standard (GSIS). 5-73
Hardware configuration. 14-27
Information Resource Dictionary System (IRDS). 5-69
Information Technology Standards Guidance (ITSG). 2-29
Interface. 5-69, 5-70
MIS. 13-30
XWindows. 14-32
Motif. 5-70, 5-73
Of excellence. 16-7
Of performance. 16-8
Of quality. 16-28
OpenLook. 5-70, 5-73
OSF/Motif. 14-32
Portable Operating System Interface for UNIX (POSIX). 2-26, 5-70, 10-24, 10-71, 14-32
Programmers Hierarchical Interactive Graphics System (PHIGS). 5-70, 5-71
Secondary. 5-70
Structured Query Language (SQL™). 5-69, 5-71, 10-71, 14-32
Transmission Control Protocol/Internet Protocol (TCP/IP). 5-71
XWindows. 5-70, 5-73, 10-71
Statement of Objectives (SOO). 11-31, 13-12
Statement of Scope. 12-26
Strassmann, Paul A. 4-20, 14-40
Strategic Defense Initiative (SDI). 1-11
Strategic Plan. 12-42
Strategic Software Management Plan. 12-25
Structure clash. 14-39
Structured analysis. 14-15
Structured Query Language (SQL™). 5-71, 10-71
Subcontractor. 12-21, 13-14
Management Plan. 13-15
Support. 4-22, 8-50, **11-3**, 11-8. *See also Continuous Acquisition and Life Cycle Support (CALS); Logistics Support Analysis (LSA)*
Code translation. 11-16
Cost drivers. 11-5
Cost of. 11-5, 11-13
Documentation
Integrated Logistics Support Plan. 12-13
Estimation of. 12-25
In the RFP. 11-29
Logistics support analysis (LSA). 11-19, 13-31
Of COTS. *See Commercial-off-the-shelf (COTS) software*
Joint Command COTS Supportability Working Group. 13-29
Planning. 11-11
Post-Deployment Software Support Concept Document (PDSSCD). 11-23
Post-Deployment Software Support Plan (PDSSP). 14-75
Problems. 11-9
Productivity. 11-5
Redocumentation. 11-16
Restructuring. 11-16
Retarget. 11-16
Reverse engineering. 11-15
Software Supportability Checklist. 11-19
Sole-source. 13-29
Strategies. 11-33
Support software. 12-31
Sustaining Base Information Services (SBIS). 5-14
Syntax. 5-7
System. 3-4
Defect. *See Defect*
Distributed. 5-60
Programming. 5-60
System Program Director (SPD). 3-16
System Requirements Specification (SRS). 8-40
System software. 12-31
System/Segment Design Document (SSDD). 15-22
System/Segment Specification (SSS). 12-24, 12-26, 12-29, 13-14, 15-68
Systems Concept Paper (SCP). 12-13
Systems engineering. 4-12, 10-21
Systems perspective. 14-5
Systems Engineering - Capability Maturity Model (SE-CMM℠). 7-26
Architecture. 7-27
Common features. 7-27
Generic practices. 7-27
Systems Engineering Management Plan (SEMP). 4-14, 12-13

T

Team
  Acquisition team. 13-11
  Building. 13-3, 13-39
  Communication. 6-9, 6-17
  Cooperation. 1-54
  Empowerment. 1-56
  Government/industry team. 14-7
  Operational Requirements Document support team. 14-21
  Operational requirements working group. 14-21
  Partnership. 13-18
  Peer inspection team. 15-44
  Process action team (PAT). 15-17, 15-29, 15-32, 15-35, 15-46
  responsibilities. 15-35
  Software requirements team. 14-21
  Source selection team. 13-12, 13-45
  Spirit. 1-57
  Teamwork. 1-54, 13-3, 13-8, 13-39, 14-7
Teamwork®/Ada. 10-52
Technical Architecture Framework for Information Management (TAFIM). *See Architecture*
Technical performance measurement (TPM). 4-14
Technical Reference Model (TRM). 2-29
Technology. 10-8, 10-75. *See also Tools*
  Assessment. 12-17
  Insertion
    Ada Technology Insertion Program. 10-71
  Modernization of. 13-21
  Strategy. 10-11
  Support programs. 10-71
  Technology Plan. 10-11
  Transition. 10-19
  Transition programs
    Advanced Computer Technology (ACT). 10-74
    Computer Resource Technology Transition (CRTT). 10-74
    Embedded Computer Resources Support Improvement Program (ESIP). 10-74
  Transitioning. 10-19, 16-14
Testing. 5-66, 14-8, 14-16, 14-50, 15-22, 15-42
  Behavioral. 14-56
  Developer testing. 14-55
  Documentation
    Final Test Report. 14-66
  Formal Qualification Test (FQT). 5-40
  Functional certification test (FCT). 14-58
  Government testing. 14-60
  Integration testing. 5-40, 14-59
  Objectives. 14-51
  Of requirements. 14-16, 14-56
  Structural. 14-56
  Systems. 14-60
  Terminology. 14-66
  Test and Evaluation Master Plan (TEMP). 3-7, 12-13
  Tools. *See Tools*
  Unit. 14-57
  Unit testing. 14-55, 14-67
    Cleanroom engineering. 15-51
    problems. 14-59
TestMate. 10-63
Theater Display Terminal. 5-30
Thor missile. 12-17
Threat. 1-28, 14-71
  System Threat Assessment Report (STAR). 14-71
Threshold. 12-46
Throughput. 4-33
Tools. 16-14
  Ada tools. 5-38
    typical Ada toolset. 10-14
  Benefits. 10-6
  Computer-aided software engineering (CASE). 4-28, 10-6, 10-21, 14-16, 14-34, 15-61
    AdaSAGE. 10-49
    cautions about. 10-18
    repository. 9-18
  Configuration management (CM) tools. 10-69
    Process Configuration Management Software (PCMS). 10-70
  Design tools
    Ada Semantic Interface Specification (ASIS). 10-73
    Common Object Request Broker Architecture (CORBA). 10-53
    SQL Ada Module Extension (SAME). 5-72
    Teamwork®. 10-52
    Universal Network Architecture Services (UNAS). 5-74, 5-78, 10-22, 10-25, 13-43, 14-22, 14-35, 14-38, 16-23

INDEX

In the RFP. 10-79
Management tools. 10-35
AdaMAT. 10-66
ProcessWeaver. 10-46, 13-42
Program Management Support System (PMSS). 10-42
Measurement/metrics tools. 10-66
AdaMAT. 10-66
AdaQuest. 10-66
Amadeus Measurement System. 10-67
SNAPSHOT. 11-18
SOftware Reliability Modeling and Analysis Tool Set (SORTS). 10-65
Planning for. 10-11
Prototyping tools
Requirements Engineering Environment (REE). 10-47
Sammi Development Kit (SDK). 10-49
Re-engineering tools. 10-64
SOftware Reliability Modeling and Analysis Tool Set (SORTS). 10-65
Reuse tools. 9-41
Selection. 6-7, 10-13
Simulation tools, hardware
Very High Speed Integrated Circuit (VHSIC) Hardware Design Language (VHDL). 14-25
Software engineering environment (SEE). 1-51, 5-38, 10-21, 10-79, 11-12
Ada. 5-38, 11-12
Ada programming support environment (APSE). 10-21, 10-71
Aeronautical Systems Center (ASC)/ Software Engineering Environment (SEE). 10-28
COHESION™. 10-28
Support tools
RECORDER. 11-18
Testing tools. 10-55, 14-64
AdaQuest. 10-58, 10-66
AdaTEST. 10-60
AdaWISE. 10-59
Analysis of Complexity Tool (ACT). 10-62
Battlemap Analysis Tool (BAT). 10-62
MathPack. 10-60
McCabe Design Complexity Tool. 10-60
McCabe Instrumentation Tool. 10-62
McCabe Slice Tool. 10-62
TestMate. 10-63
Tool support centers. 10-75
Software Engineering Institute (SEI). 10-76
Software Technology Support Center (STSC). 10-75
Standard Systems Center (SSC). 10-76
Total quality management (TQM). 15-4, 16-5
Total Quality Master Plan (TQMP) (DoD). 15-4
Track record. *See Contractor*
Training. 2-36, 5-15, 6-7, 6-17, 10-38, 13-25, **15-63**
Ada. 5-38, 15-63
Central Archive for Reusable Defense Software (CARDS). 9-38
For peer inspections. 15-48
Just-in-Time. 15-65
Peer inspection. 15-48
Plan. 15-65
Transmission Control Protocol/Internet Protocol (TCP/IT). 5-71
Trusted - Software Capability Evaluation (T-SCE). 7-26
Trusted Computer System Evaluation Criteria (TCSEC). 14-74
Trusted Software Development Methodology (TSDM). 7-26
TYCOM Readiness Management System (TRMS). 5-30
Tyndall Range Control System (RCS). 5-53
Type. 5-57, 5-60
Type Commander Readiness Management System (TCRMS). 10-51

U

Understandability. 4-33
Uniformity. 4-36
Unit. 14-57
Testing. *See Testing*
US Army Information Systems Software Development Center - Lee (USAISSDCL). 7-32
US Strategic Command (USSTRATCOM). 7-35, 11-18
Usability. 14-63
Usage probability distribution. 15-52
User. 13-10, 14-22
Documentation. 14-78

Involvement. 1-23, 3-9, 10-7, 14-41, 15-60
Satisfaction. 14-31
USS Seawolf (SSN 21) Submarine. 2-7, 5-22
AN/BSY-2 Project. 5-22, 13-29
lessons-learned. 13-44

V

Version control. 14-20
Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). 10-72, 14-25
Vessey, GEN John W. 1-26, 12-10
Vulnerability. 14-71

W

Walkthrough. 15-41
War Planning Force Application. 11-18
Warner Robbins Air Logistics Center (WR-ALC). 15-11
Weapon system. 2-9
Weapon systems software. 2-9
Command, control, and communications (C3). 2-9, 2-11
Intelligence (C3I). 2-11
Welch, Secretary John, Jr. 1-34
Work breakdown structure (WBS). 12-24, 12-26, 12-29, 16-18
Contract. 13-13
Element. 12-30, 12-31
Interrelationships. 12-29
Prime mission product summary. 12-30
Software contract. 12-33
Software project. 12-35
Software project summary. 12-31
Summary. 13-13
Work package. 12-30, 15-16
Work plan. 6-7

X

XWindows. 5-70, 5-73, 10-71

Y

Yates, Gen Ronald W. 5-5

Version 2.0

DIRECT COMMENTS, QUESTIONS, OR REQUESTS FOR ADDITIONAL COPIES TO:

Software Technology Support Center
Ogden ALC/TISE
7278 Fourth Street
Hill AFB, Utah 84056-5205
(801) 777-8045
E-mail: custserv@software.hill.af.mil

Blank page.