

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
APR 3 1985
S B D

AD-A152 067

THESIS

A FRAMEWORK FOR SOFTWARE DEVELOPMENT

by

Eric C. Hughlett

September 1984

Thesis Advisor:

Dean Guyer

Approved for public release; distribution unlimited

DTIC FILE COPY

85 03 15 033

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Framework for Software Development		5. TYPE OF REPORT & PERIOD COVERED Master's Degree September, 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eric C. Hughlett		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE September 1984
		13. NUMBER OF PAGES 101
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) metrics, standardization, Ada, maintenance, quality assurance, stars		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) All sectors of society are confronted with what has been termed the "software crisis". As the world's largest single buyer of software, the Department of Defense has undertaken major software initiatives to ameliorate software-related problems associated with major computer systems acquisition. This thesis provides an overview of common problems in both embedded and administrative software development and acquisition. It defines quality software in terms of its characteristics, and provides the project manager/acquisition (Continued)		

ABSTRACT (Continued)

agency with a set of accepted controls to assure that quality is built in to software for improved maintainability. The difficulties and limitations of providing accurate estimates in software development are discussed in terms of software metrics. A number of DoD current and future standardization efforts, including the Army's development of a Military Computer Family (MCF), Ada, and the STARS initiative.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution unlimited.

A Framework
for
Software Development

by

Eric C. Hughlett
Lieutenant Commander, United States Navy
E.S.B.A., Appalachian State University, 1975

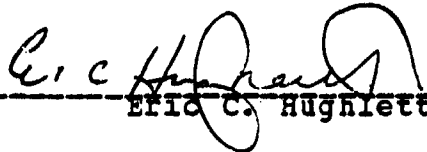
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
September, 1984

Author:


Eric C. Hughlett

Approved by:

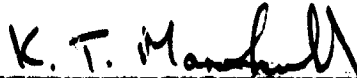


D. C. Guyer, Thesis Advisor


N. R. Lyons, Co-Advisor



W. R. Greer, Jr., Chairman,
Department of Administrative Sciences



K. T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

All sectors of society are confronted with what has been termed the "software crisis." As the world's largest single buyer of software, the Department of Defense has undertaken major software initiatives to ameliorate software-related problems associated with major computer systems acquisition. This thesis provides an overview of common problems in both embedded and administrative software development and acquisition. It defines quality software in terms of its characteristics, and provides the project manager/acquisition agency with a set of accepted controls to assure that quality is built in to software for improved maintainability. The difficulties and limitations of providing accurate estimates in software development are discussed in terms of software metrics. A number of DoD current and future standardization efforts are discussed, including the Army's development of a Military Computer Family (MCF), Ada, and the STARS initiative. Additional keywords: theses,

Quality assurance, specifications
STARS computer program, STARS (Software
Technology for Adaptable Reliable Systems),
productivity.

TABLE OF CONTENTS

I.	INTRODUCTION	10
	A. BACKGROUND	10
	B. THE COST OF SOFTWARE IN DOD	11
	C. PURPOSE AND APPROACH	14
II.	THE SOFTWARE CRISIS	16
	A. PROBLEMS IN SOFTWARE ACQUISITION	16
	1. A GAO Report	16
	2. The Multi-source Unified Data Distribution (MUDD) Report	23
	3. DoD Weapon System Software Study	24
III.	MEASURES OF CONTROL	26
	A. BACKGROUND	26
	B. CAUSES FOR POOR SOFTWARE ESTIMATING	27
	1. Lack of Estimating Expertise.	27
	2. Biases in Estimating	27
	3. Poor Understanding of What Estimate Means	28
	4. Estimates as Basis for Incentives	28
	C. SOFTWARE METRICS	29
	1. Halstead's Software Science	29
	2. McCabes' Complexity Measure	31
	D. SOFTWARE COSTING	32
	1. Analogy	32
	2. Decomposition	33
	3. Parametric Models	33
	E. CHAPTER SUMMARY	35

IV.	QUALITY SOFTWARE	37
A.	BACKGROUND	37
B.	DEFINING SOFTWARE QUALITY	38
C.	CHARACTERISTIC OF SOFTWARE QUALITY	39
D.	QUALITY ASSURANCE	41
E.	IMPLEMENTATION OF A SOFTWARE QUALITY ASSURANCE PROGRAM	44
	1. Procuring Agency Evaluation	45
	2. Design Inspection	46
	3. Code Inspection	48
	4. Test	49
	5. Library Controls	50
F.	PARTING COMMENTS	50
V.	SOFTWARE MAINTENANCE	51
A.	CATEGORIZATION OF MAINTENANCE ACTIVITIES	51
B.	TANGIBLE MAINTENANCE COST	52
C.	VARIABLES AFFECTING MAINTENANCE COSTS	55
D.	INTANGIBLE MAINTENANCE COSTS	56
E.	BUILDING MAINTAINABLE SOFTWARE	56
	1. Structured Methodology	57
	2. Structured Analysis	58
	3. Structured Design	59
	4. Structured Programming	60
	5. Program Design Language	60
F.	PARTING COMMENTS	61
VI.	DOD STANDARDIZATION AND SPECIFICATIONS	62
A.	INTRODUCTION	62
B.	SPECIFIC INITIATIVES	62
	1. Military Computer Family	63
	2. Ada	64
	3. Joint Logistics Commanders Workshop	66

VII.	STARS	69
A.	OVERVIEW OF STARS	69
B.	OBJECTIVES	71
C.	ORGANIZATION	73
D.	EFFECTIVE MEASUREMENTS	73
E.	PROJECT MANAGEMENT	74
F.	IMPROVING PERSONNEL RESOURCES	76
	1. Key Population Assessment	76
	2. Career Structures and Incentives	77
	3. Exchange Programs	77
	4. Other Educational Subtasks	77
G.	IMPROVING PROCESSING TECHNOLOGY	78
H.	INCREASING USE OF PROCESSING TECHNOLOGIES	79
	1. Improve Business Practices	79
	2. Improve Tool Usability	80
I.	CONCLUSION	81
VIII.	CONCLUSIONS AND RECOMMENDATIONS	82
	APPENDIX A: GLOSSARY OF SOFTWARE QUALITY ATTRIBUTES	85
	APPENDIX B: HALSTEAD AND MCCABE'S SOFTWARE METRICS	87
	A. HALSTEAD'S SOFTWARE SCIENCE	87
	B. MCCABE'S COMPLEXITY MEASURE	92
	APPENDIX C: STRUCTURED METHODOLOGIES	94
	A. STRUCTURED ANALYSIS	94
	B. STRUCTURED DESIGN	95
	C. STRUCTURED PROGRAMMING	96
	D. PROGRAM DESIGN LANGUAGE	96
	LIST OF REFERENCES	97
	INITIAL DISTRIBUTION LIST	101

LIST OF TABLES

1.	Cost Trends: Hardware versus Software	14
2.	Language Level Values	31
3.	Comparison of Cost Estimating Methods	34
4.	Evaluation Factors in Bidder Responses	46
5.	Ada Specifications	65
6.	A Sorting Subroutine	88
7.	Operator Count	88
8.	Operand Count	89

LIST OF FIGURES

2.1	Value of Delivered Software	22
4.1	Characteristics Tree	40
4.2	Cost Impact of Changes	43
4.3	Basic Code Structures	48
5.1	Maintenance Cost as Percentage of Budget	53
5.2	Life Cycle Maintenance Costs	54
E.1	Control Flow Graph Complexity	92

I. INTRODUCTION

A. BACKGROUND

"And a good south wind sprung up behind;
The Albatross did follow,
And every day, for food or play,
Came to the Mariner's hollo!

God save thee, ancient Mariner!
From the fiends that plague thee thus!--
Why looke'st thou so?--With my cross-bow
I shot the Albatross." [The Ancient Mariner, pt. i]

The albatross around today's program manager neck is often the software subcomponent of major system acquisitions. Cost overruns, schedule slippages, and loss of program control have been the penance for those project managers who have failed to provide for software with the same intensive and continuing management typically rendered its hardware counterpart.

Software is an intangible product that defies description in an engineering sense. Only a few software products have ever started off with clear, unambiguous, and definitive requirement specification. Schedules and costs are often dictated by the system acquisition milestones and reviews, and not necessarily associated with the phased software development methodologies advocated by what has been termed "software engineering¹". Many of the specific problems that surround software development and acquisition will be discussed in detail in the next chapter.

¹The key objectives of software engineering are (1) a well-defined methodology that addresses all phases of the software life cycle, (2) an established set of software components to document and show traceability from one development step to the next, and (3) a set of predictable milestones that can be reviewed as needed [Ref. 1: p 15].

In the majority of guidance and managerial principles available to assist the program manager are directed at the hardware end. Software is the "new kid on the block." It is that part of the system that is seldom understood and often mismanaged during system acquisitions. Computer hardware, on the other hand, has undergone remarkable improvements in function, size, performance, and relative cost. Several hardware generations have emerged in the course of a single human generation. Yet, software has experienced more noticeable growing pain. The gap between hardware-- and software technology widens.

B. THE COST OF SOFTWARE IN DOD

There are two general classifications of software within DoD. The first of these is that of the more-traditional, administrative type of software used in business applications. This type of software is typically supported by commercially available computer that can support a variety of applications, i.e., Automatic Data Processing (ADP, systems. The second classification is embedded software. Embedded software is normally designed to operate as an integral part of non-ADP systems, such as DoD tactical systems. The most significant difference between these two classifications of software rest not in the development and maintenance practices, but rather in the frame work in which they are each procured.

The procurement authority for Automatic Data Processing Equipment (ADPE) and its supporting software and services is vested in the General Services Administration (GSA), as directed by Public Law 89-306, 40 USC 759, the "Brooks Bill." Within DoD, ADPE is under the purview of the Assistant Secretary of Defense (Comptroller). Weapon system software is under the cognizance of the Office of the

Undersecretary of Defense (Research and Engineering).² Although there is a distinct dichotomy of cognizant organizational structures regulating the acquisition of ADP and non-ADP software, the managerial and software engineering principles which govern each step of the software life cycle are, in fact, quite similar. Therefore, the common set of tools, methods, and methodologies advocated in this thesis apply to both ADP and non-ADP software.

In writing this thesis, it was noted that the majority of available DoD guidance for the control and acquisition of software projects was in support of tactical systems, with the vast majority being authored for the United States Air Force. This is not surprising since it has been estimated [Ref. 2 : p. 7] that of the \$12 billion that DoD will spend for software in 1985, over \$10 billion will be for embedded software, with the U. S. Air Force accounting for approximately half of the expenditures.

Not only does embedded software represent the largest component of total software costs in DoD, it is also plagued to a proportional degree with many of the software-related problems, which M.M. Lehman so aptly describes as

"a motley collection of relatively isolated methodologies and techniques, associated through an experience-based, but otherwise arbitrary sequence of much-discussed process phases". [Ref. 3 : p. 3]

At this point, it is important to recognize some of the some of the program characteristics that add to the complexities of DoD's embedded software. These include: [Ref. 4 : p. 77]

²Due in large part to the provisions set forth in the subsequent Warner Amendment, the policies and procedures set forth in the Brooks Bill does not extend to the tactical software used in DoD weapons systems.

- program size--often in excess of a million lines of code.
- real-time operating requirements requiring response time in milliseconds.
- programs must be flexible to accommodate changes in system evolution over an expected useful-life often in excess of twenty years.
- guaranteed reliability due to the tight (and many times life-dependent) coupling between the system and its user of the population that the system is designed to protect.
- programs are part of the universe which they model and control.

As compared to other software applications, such as ADP or administrative computing, DoD mission-critical software is more complex, less understood, more unstable, and must operate in extreme environmental conditions. Yet it is essential that DoD software be reliable, adaptable, and affordable. To achieve these objectives, many problems, of both a technical and managerial nature, must be overcome. Symptoms of these problems include slippages in weapon delivery schedules, system failures, overbudget programs, and inflexible systems will be discussed at further lengths in Chapter II.

DoD is recognized as the world's largest buyer of software. Based on various estimates in recent literature, it is calculated that DoD will spend approximately \$12 billion for software in 1985 [Ref. 2 : p. 5]. Table 1 illustrates the percentage of total computing system costs of hardware as compared to software for all of DoD computing systems. Software cost reflect all aspects of the software life cycle, including: design, development, testing, operations, and maintenance. The ratio of hardware to software has reversed itself from 4:1 to that what is expected to approach 1:9 next year. [Ref. 2 : pp. 5 - 6].

The high cost of acquiring software has naturally caused concern in both DoD and the Congress. Literature

TABLE 1
Cost Trends: Hardware versus Software

	(percentage of total cost)			
	1955	1970	1979	1985 (Estimate)
Hardware	83	45	25	10
Software	17	55	75	90

abounds with studies and recommendations related to software development in DoD. There is not a shortage of sage advice. The need for improved managerial controls and software development practices has been recognized.

C. PURPOSE AND APPROACH

A major goal of this thesis is to present a consolidated review of major DoD efforts aimed at reducing software-related problems. Both management and technical issues will be addressed. This thesis makes no pretense that it provides the program manager with all of the technical background and controls needed to assure the timely delivery of quality software within budget. Rather, it focuses on key and "high payoff" issues involved in managing the acquisition and development of software. This thesis also addresses several DoD initiatives which promise to significantly alter the framework in which software is developed and maintained.

Chapter II identifies many common problems associated with contracting for general computer software by Federal agencies. It also identifies major contributing factors to DoD weapon system software problems. A common denominator in the formulation of the many software problems is the lack of estimating expertise by which program measurements can be

defined. Chapter III discusses software metrics for defining quantifiable measurements.

The delivery of good software is an implicit, but most elusive, goal in software acquisition. Chapter IV defines "good software" through a set of quality software characteristics. It also provides a series of controls to be utilized in the implementation of a quality assurance program. Major dividends from quality software are realized during the post-development phase of software, the maintenance phase. Chapter V analyzes the tangible and intangible costs of software maintenance, and addresses a number of software engineering principles through which the costs of maintenance can be greatly reduced.

Chapter VI and VII review a number of software and computer-technology standardization initiatives to undertaken within DoD. Perhaps the most significant of these initiatives is the STARS (Software Technology for Adaptable, Reliable Systems) program.

Finally, Chapter VII provides this thesis' conclusions and recommendations.

II. THE SOFTWARE CRISIS

"The problem of the 1970's was to reduce the cost of the electronic functions needed to store and process data....

"The problem of the 1980's is different. Now we must reduce the cost of electronic solutions; that is, reducing the cost you incur in using our devices to build a product. Solving this problem will require a shift from the component integration of the 1970's to concentration of system level integration in the 1980's.

"We can now put about the power of a mainframe CPU on a single chip. This buys you nothing as a customer, however, unless you can use that power. Hardware is computing potential; it must be harnessed and driven by software to be useful." [Ref. 1 : p.22]

The preceding statement was made by the president of one of the largest manufacturers of computer hardware. It succinctly summarizes the shift in technological emphasis from hardware to software.

A. PROBLEMS IN SOFTWARE ACQUISITION

To the casual observer, the successful management of a software development project may seem a simple process. All that is needed are (1) well-defined requirements, (2) realistic cost and schedule estimates, and (3) the right quantity of personnel and hardware at the right time. In actuality, each of these elements seldom, if ever, happens by themselves, much less together.

The management of software development projects have historically been plagued by a myriad of problems, both in the private and government sectors.

1. A GAO Report

Recently GAO reported to Congress [Ref. 5 : pp. 1 - 84] a number of problems that Federal agencies have

encountered in contracting for computer software as an alternative to in-house development. Means for improving these deficiencies were also recommended.

a. Scope of the GAO Report

GAO sent questionnaires to 163 software contracting firms and 113 Federal project offices that had experience with software development projects. The purpose of the questionnaires was to attempt to identify common problems in software development contractual process and what, through hindsight, might have been done to prevent or improve development efforts.

GAO examined nine cases of software development in detail, some of which had attracted GAO attention because they were known failures. Only one of these nine cases yielded a software product that could be used as delivered.

The actual combined total development cost and time for the nine cases almost doubled the estimates of \$3.7 million and 10.8 years.

b. Common Causes of Software Contracting

The nine cases that were studied in detail illustrated many of the same causes of difficulties that respondents to the GAO's questionnaires had identified. The most significant of these findings will now be described:

- Federal agencies contract for software development with little specific guidance.

Guidelines for software development promulgated by central agencies are primarily aimed at the technical aspects of software development. Very little guidance is provided in support of the contractual process.

Basic responsibilities of the central agencies are set forth in the Brooks Act, Public Law 89-306. The

Office of Management and Budget (OMB) is prescribed general oversight of Automatic Data Processing (ADP) activities. Much of this responsibility has been delegated to the General Services Administration (GSA) and to the National Bureau of Standards (NBS). GSA is delegated the responsibility for ensuring cost effectiveness in the selection, acquisition, and utilization of ADP resources. GSA's guidance for the management of ADP resources is contained in subpart 101-32 of the Federal Property Management Regulations.³ Policies addressing the procurement of and contracting for commercially available software is provided in Federal Procurement Regulation 1-4.11. GAO's review of both of these documents revealed that there is very little actual guidance directed at the specific contractual management for engaging in custom software development.

Although NBS is tasked with developing technical standards and guidelines, OMB has indicated that NBS is also responsible for investigating and assisting in software system developments. Although NBS representatives advised GAO that their responsibilities involved managerial and contractual activities for system development, NBS' emphasis has been, and will continue to be, on the technical aspects of system development, such as the standardization of government-used Higher Order Languages.

-- Agencies overestimate the stage of their own preliminary work before they contract.

GAO found two primary reasons why agencies contract out for software development instead of doing it in-house. The first is that many of the agencies lack sufficient quantities of, or properly skilled, personnel to do

³As of 1 April, 1984, DoD regulations concerning the acquisition of ADPE resources and services previously contained in the Defense Acquisition Regulations (DAR) have been replaced by DoD supplements to the Federal Acquisition Regulations (FAR); specifically, Subchapter H, Part 70 of the DFAR.

the work. Secondly, the software is often needed sooner than it can be produced in-house. Often the initial steps of software development, such as requirement analysis, are started in-house prior to contracting out for the continued development of required software. Two common problems have been observed in this context. First, the agency may overestimate the amount of work already achieved in-house

secondly, the agency's preliminary work that is turned over to the contractor may be inadequate requiring that it be done again by the contractor.

Overestimating the stage of software development before releasing it to a contractor is likely to result in additional costs to the extent that any cost benefits that might have been gained from the development project are forfeited. It is critical that precise methods for measuring preliminary in-house work be used in order to achieve realistic cost and time estimates. An accurate identification of the stage of system development is vital in order to properly determine the type of contract to be utilized. If, for example, the agency has completed all the preliminary development stages required prior to the commencement of coding, then a firm-fixed price contract for the coding effort might be the most suitable. If, on the other hand, a systems detailed design has not been completed by the agency prior to entering into a contractual agreement, then a phased, cost-plus- fixed-fee type contract would likely be more suitable since the exact scope of future efforts is not yet known.

If agency work that is passed on to the contractor is later found to be inadequate, or less than originally estimated, much of the work may have to be redone by the contractor. In doing so, there often is a tendency to attempt to save as much of the original work as possible in order to remain within the cost and time ceiling mandated by

the contract. This is likely to compromise the design of the new system, resulting in a less efficient system that mandates higher operating and maintenance cost for the remainder of its life cycle.

- Contracts fail to stipulate what constitutes satisfactory performance.

Failing to stipulate what constitutes satisfactory performance by the contractor makes it difficult, if not impossible, to claim poor contract performance. Furthermore, it reduces the probability of a satisfactory end-product. Many disputes over contractor performance could be avoided if adequate system specifications and testing criteria are identified in the contract.

Other general requirements and constraints that can usually be identified at the start of a software project criteria for software expandability, documentation standards, maximum computer resources allowable, maintenance, and program transfer capabilities.

- Agencies quickly overcommit themselves, and fail to adhere to strict phasing to control contractors.

Phasing divides the development effort into logical and manageable work phases. One of the most effective controls available to an agency is in the contractual identification of phases, coupled with mandatory agency review and approval following each phase as a precondition to the contractor's continuation of subsequent phases. Other advantages associated with phasing include:

- Identification of milestones and timetables to monitor the progress of the project, allowing for the initiation of corrective actions in a timely fashion.
- Systematic and orderly development of software.
- Control of funds based upon quality and acceptability of contractor's work.
- Increased assurance that should development efforts are being used.
- Improved communication between the agency and the contractor leading to the increased probability

that the contractor fully understands the agency's requirements.

- Completed phases provide an adequate base upon which subsequent phases can be built.
- Lack of agency management during contract execution.

An excessive number of system changes were requested by the agencies in the cases studied by GAO. These agency-initiated changes ranged in scope from minor requirement adjustments to re-design of the entire system. Many of these changes requested and made during the latter phases of development and contributed significantly to cost and schedule overruns.

Project managers should be aware of the need for a well-defined problem statement and the undermining effects that changes have on software development. Changes, as compared to the original requirement specification, are not usually as thoroughly researched and may cause unforeseen and rippling effects on other parts of the system. The systematic and logical flow of contract phasing may be lost due to the need to modify work that has already been completed and approved, obscuring the visibility of the project's status. Furthermore, excessive changes make it difficult to hold the contractor accountable for the initial terms of the contract.

- Agencies to not adequately inspect and test software.

As depicted in figure 2.1, most of the software delivered in the cases studied was of poor quality. Reasons for this poor quality was evidenced in all phases of development. Quality assurance must be tied directly to the contractual process. Higher quality software can be obtained if the contractor maintains quality assurance functions in a number of software development areas. Specific examples of these areas include configuration management, testing, program design, documentation, and working tasks.

The latter of these area, working tasks, is a means for assuring that procedures are in affect for subdividing the total work effort into segments and assigning responsibility for the initiation and completion of work.

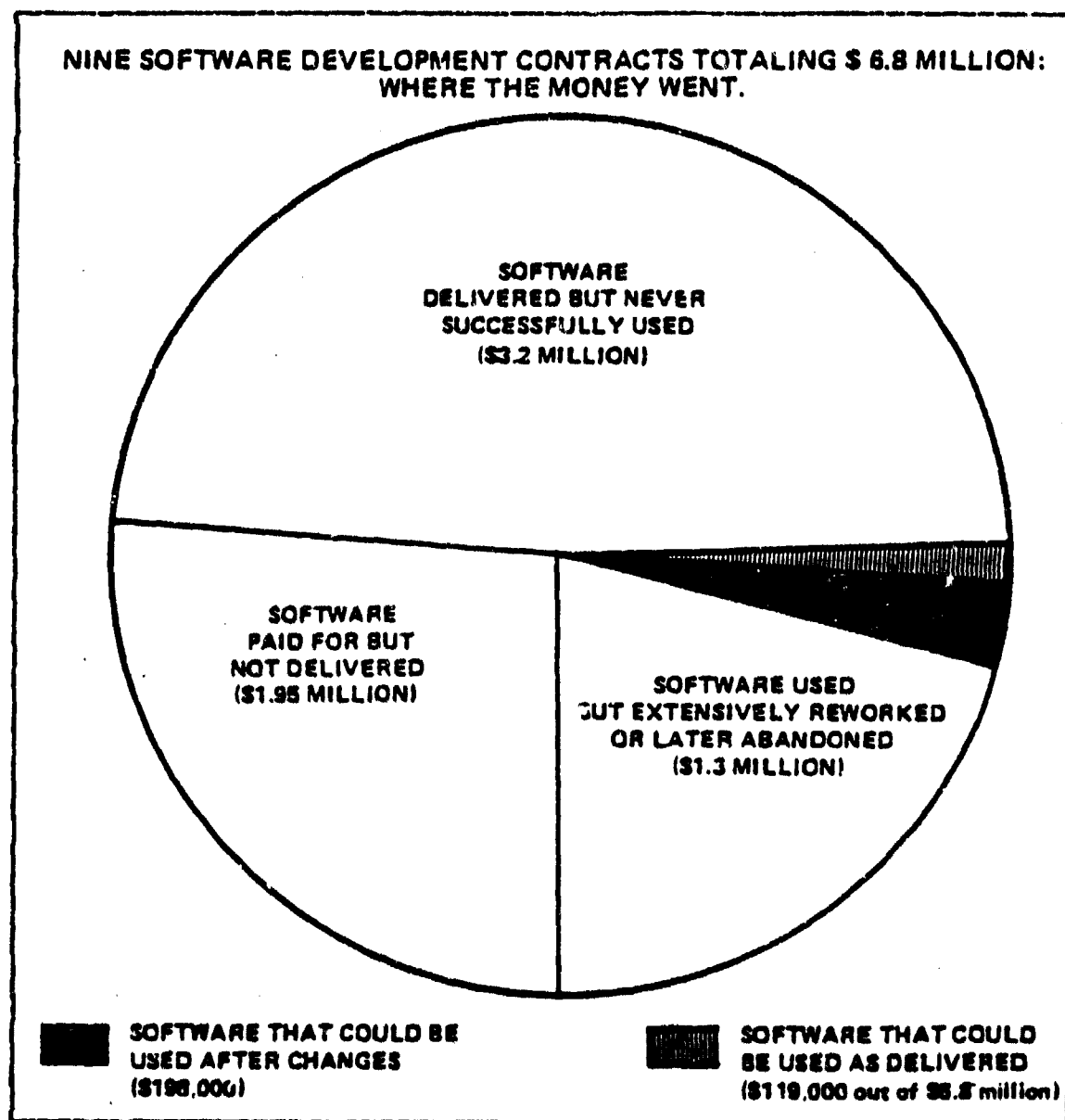


Figure 2.1 Value of Delivered Software

The GAO report concluded by stating the need for improvements in contracting for custom software development. Recommendations were made aimed primarily at GSA and NBS for both improvements in both procurement and technical areas.

GAO further recommended that GSA and NBS work together in designing model contracts of various types. These contracts would have sample clauses for covering the withholding of payments, testing, etc.. Agencies would use these samples to extract those clauses which best fit their particular requirements.

The last recommendation that GAO made was that Federal agencies that extensively contract for software development "should train project managers in appropriate software, contracting, and management skills." [Ref. 5 : p. 29]

2. The Multi-source Unified Data Distribution (MUDD) Report

The MUDD Report [Ref. 6 : pp. 1 - 28] should be considered "required reading" by all present and future project managers overseeing software development. It is a case study of Navy software development practices. The report is based on over 30 interviews with those responsible for the development of Navy systems. The year-long study uses the development of the fictional MUDD system under development to mirror many of the requirements of Navy tactical systems either in operation or under development. It chronicles and analyzes the decisions made on the software development effort. The MUDD Report concludes with a set of recommendations to Navy program managers for avoiding the pitfalls described in the report.

The issues brought to light in the MUDD report are germane to those problem areas found in large and complex system development efforts which typify many DoD programs.

An adequate summation can not be given of the MUDD report which can do it justice. It should be read in its entirety for a full appreciation of Weiss' recommendations which are directed at problem areas that infest the fictional MUDD system development. Most of the recommendations center on various types of interfaces, such as the interface between the Navy and contractor, interfaces between people, interfaces between and within systems, and interfaces within the Navy.

3. DoD Weapon System Software Study

The John Hopkins University Applied Physics Laboratory (APL/JHU), in conjunction with the MITRE Corporation, conducted an extensive study of the management of weapon systems software under the auspices of the Office of the Secretary of Defense [Ref. 27]. The MITRE and APL study team reviewed the findings of ten previous DoD-sponsored studies relating to software. The MITRE Corporation concluded that the "major contributing factor to weapon system computer software problems was a lack of discipline and engineering rigor applied consistently to the software acquisition activities." [Ref. 7 : p. 50] Other, more specific, findings of included in the MITRE/APL study included the following: [Ref. 7 : pp. 50 -51]

- Frequent contributors to software cost and schedule growth include: (a) poorly formulated initial software requirements; (b) changing requirements and requirement growth during the development phases; (c) false starts and need to educate involved organizations before useful output can be obtained; (d) inefficient use (proliferation) of already existing resources; (e) inefficient testing and verification tools and methods; and (f) improper use of standards and guidance documents in specific procurements.
- There is a general need for better identification of software terms, measures of software qualities, and the methods for measuring them.
- Software technology improvements particularly aimed at developing a software engineering discipline are being made by industry, academia, and the services but require application to real military systems (in

addition to laboratory or experimental systems) for evaluation and confirmation.

The study resulted in a series of 17 recommendations, each of which was directed at a specific problem area. A sample of these recommendations included: [Ref. 7 : pp. 50 - 51]

- Specify that major computer software involved in weapon system development be designated "configuration items" and be deliverable during full-scale development.
- Use top-down design, i.e. specify the use of modular software architecture and an orderly, phased design approach that defines the higher levels of the program and then progresses to design and test successively lower levels.
- Require the contractor to apply a highly disciplined set of engineering practices.
- Establish a common set of requirements and criteria to be applied...by all services.
- Prepare a series of handbooks of guides covering important aspects of software acquisition.

While extensive progress has been made in DoD toward addressing many of the problem areas noted in the preceding studies, much work remains to be completed. Specific corrective actions that have been adopted, or which are presently in the formulation stages, will be covered in this thesis, particularly in the chapters addressing standardization and the STARS software initiative.

III. MEASURES OF CONTROL

A. BACKGROUND

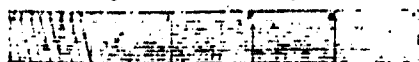
"You can't control what you can't measure." [Ref. 8 : P. 3] A disparity exists between the software manager's definition of what constitutes a project's success as compared to the user's perception of the same project. With software projects resulting in utter failures or cost overruns of two, to three, times the original estimate, software managers often consider their projects a success if overruns are kept below 30% and when "most" of the delivered lines of code are considered "usable" by the end-user.

DeMarco [Ref. 8 : p. 4] writes that many projects fail eventhough the project managers have excelled in those characteristics that he associates with good management. These characteristics include:

- project staff members that are highly motivated
- clear understanding of the issues
- adequate grasp of relevant technologies
- evident capability in the political sphere

Demarco attributes the failure of these project managers to the fact that they have simply failed to meet the original expectations of the project. He is convinced that in often it is not the fault of the project team, but rather the fault of "inflated and unreasonable expectations." When expectations exceed what can be delivered, the project is doomed to failure.

SANTEE and COOPER RIVERS
from an Actual Survey
By **HENRY MOUZON** Esq
1771
A SCALE OF MILES AND CHAINS
69½ Miles to a Degree



CONFIDENTIAL
BY J. R. GALLARD

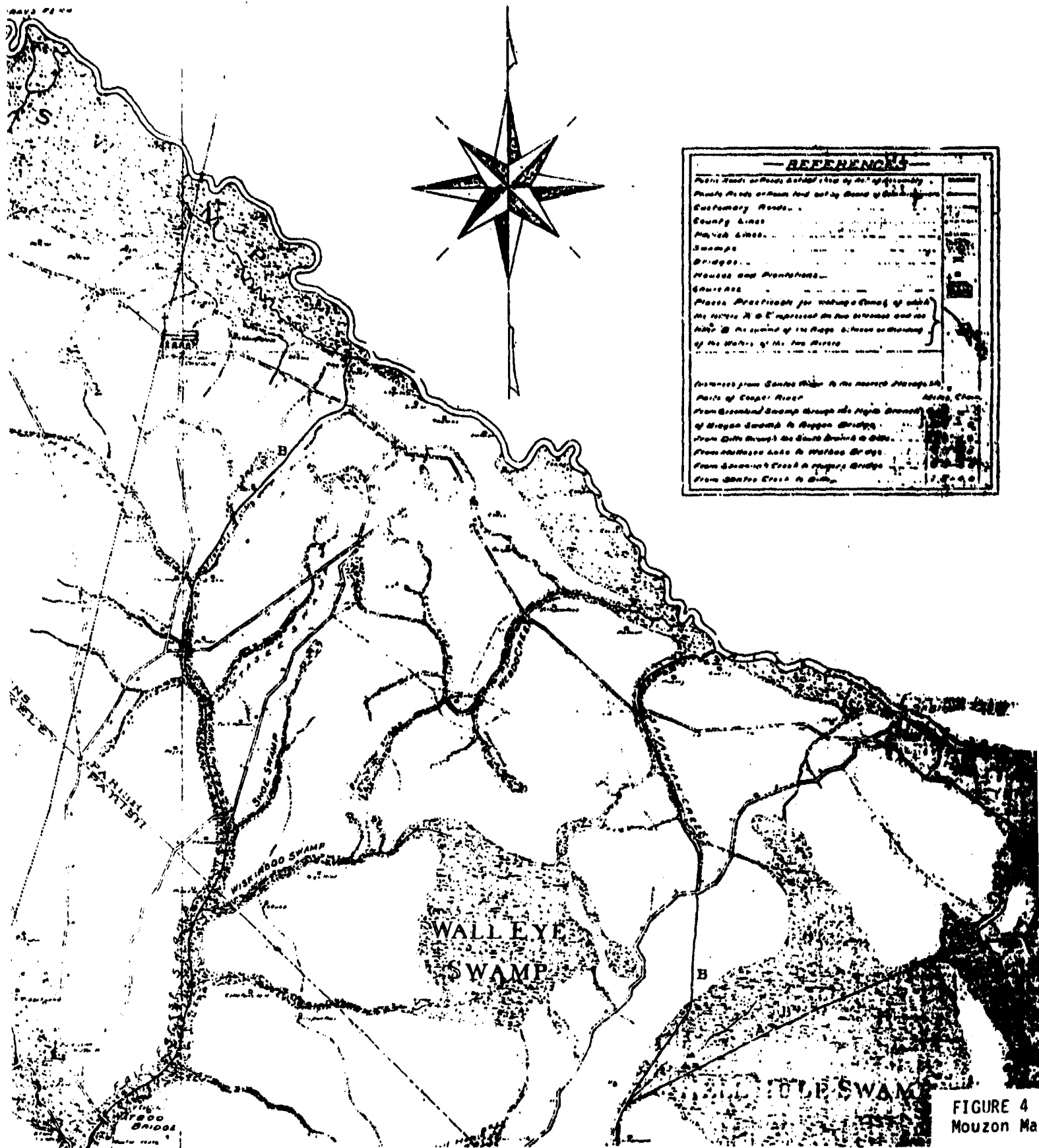


FIGURE 4
Mouzon Ma

B. CAUSES FOR POOR SOFTWARE ESTIMATING

Estimating is at the core of the difficulties surrounding software projects. Feedback is essential for control. Feedback provides a basis for comparing the actual project's progress against original expectations. These expectations were formulated on estimates. Main causes of poor software estimating are as follows: [Ref. 8 : pp. 9 - 17]

1. Lack of Estimating Expertise.

The average software manager will typically rate himself/herself well below average as an estimator. The underlying reason for this is simply lack of estimating practice or experience.

The amount of actual estimating that a typical software manager is involved in will normally take up less than 3% of his/her time. Most software projects may call for estimates at their beginning, and maybe once-a-month or prior to management review thereafter.

2. Biases in Estimating

Personal biases create a strong tendency to underestimate one's own potentials. However, when objectively estimating another's potential, then most of these biases are minimized. DeMarco suggests an obvious approach to avoid this phenomenon by stating

"Whoever does the estimating for a project must be someone whose entire ego involvement is in the quality of the estimate, rather than in the project itself...."
[Ref. 8]

3. Poor Understanding of What Estimate Means

At the very heart of probability theory is the estimation of "odds" of the occurrence of a certain event. Yet software estimates are often void of any explicit probabilistic assessment which may govern them. This observation is closely linked to preceding subsection on the personal biases involved in estimating. Should a software manager be asked the probability of finishing a project, say, 20% later than s/he originally estimated, an answer (right or wrong) will freely be given. On the other hand, should the same person be asked the probability of completing the project earlier than originally estimated, the estimator will likely give it a zero probability. This represents what DeMarco [Ref. 8 : p. 14] terms

Default Definition of "Estimate"

An "estimate" is the most optimistic prediction that has a non-zero probability of coming true.

Instead, DeMarco [Ref. 8 : p. 22] proposes that an estimate should be defined as "a prediction that is equally likely to be above or below the actual result." This definition, by itself, does not solve the estimating problem. It does, however, offer a basis from which to start examining measures and other components of estimates which will be covered in this chapter.

4. Estimates as Basis for Incentives

Often estimates are used to establish goals for performance. When used in this manner, the software manager is likely to establish estimates on previously established goals. To serve as a motivational tools for the development staff, the goals are set at unattainable levels.

Many managers view goal-oriented estimates as the supreme motivational mechanism for their overly-optimistic development staff whose self-esteem is placed on the line in the pursuit of unachievable goals. As the development staff is driven toward the completion deadline, the ultimate victim of this motivational strategy is the quality of the finished product itself.

C. SOFTWARE METRICS

The first part of this chapter has done little more than point out many of the ill-fated approaches which have traditionally been used to control software. These approaches were principally qualitative in nature, having no formal mathematical basis. Yet, intuitively, a direct relationship exists between software quality and quantitative software characteristics such as modularity, size, and logical paths. As such, software metrics have been advocated by many authors as a preferred means for deriving inputs for the estimating process.

This section will examine two of the most popular theories in software metrics that have grown out of the formative years of software engineering: (1) Halstead's Software Science Theory, and (2) McCabe's Complexity Measure. Appendix B provides sample algorithms and respective formulas for each of these two theories.

1. Halstead's Software Science

The first set of metrics to be reviewed were developed by Maurice K. Halstead [Ref. 9]. Instead of using "lines of code" (LOC's) to describe the size of a module, Halstead breaks each line down into a series, or group, of symbols. Each of these symbols can be classified as either an "operator" or as an "operand." An operand is a single

symbol or group of symbols that identifies the constants or variables that the module uses to implement its algorithm. An operator is a single symbol or group of symbols which affects the value of the operand. Operators also impact the sequence in which the operation takes place.

Criticisms concerning Halstead's theory of measuring through the use of operators and operands were quickly registered. The majority of Halstead's work evolved around algorithms drawn from Algol and FORTRAN. In other algorithmic languages, the definitions of operands and operators are not nearly as clear. Halstead also omitted declarations and comments from his calculations--a significant portion of the widely-used COBOL language. Other studies, however, have shown that the additional declarations and comments actually brought the estimated program length closer in line with the actual, completed program. In any case, it is important to identify the operands and operators of an algorithmic language to establish consistency. This function can often be determined by a compiler, through which the operators and operands are explicitly defined in the final machine language product. Questions abound on the derivation of Halstead's formulas. The validity of his experiments has been questioned because of the small sample size, the small size of the programs used, and the subjects used were college students vice experienced programmers.

Halstead proposes that each language can be categorized by a language level, L , which will vary among languages. The variances are closely linked to the level of abstraction by which a procedure can be specified. Halstead assigns a constant language level for a particular language, which is in contrast by to the recent works that show that language level is a resulting product of both the language and the programmer. Table 2 provide the language levels values that have been empirically derived for five common languages [Ref. 1: p. 166].

TABLE 2
Language Level Values

Language:	Mean 1
English pose	2.16
PL/I	1.53
ALGOL/66	2.12
FORTRAN	1.14
Assembler	0.88

2. McCabes' Complexity Measure

In his article, [Ref. 12 : PP. 308 -320] McCabe proposes a complexity measure of software which is based upon the control flow representation of a program. Through the analysis of several FORTRAN programs, he illustrates a high correlation between the intuitive complexity of a program and his proposed graph-theoretic complexity measure.

McCabe's software complexity measure is preported to measure and control the number of paths through a program. The primary difficulty in this regard is manifested through backward branches which may possibly result in an infinite number of paths during program execution. Consequently, using a path count to measure program complexity is impractical. However, the complexity measure can be defined in terms of basic paths, that when taken in combination, will measure every possible path.

As compared to Halstead's metrics, McCabe's complexity measure can be applied during the earlier stages of software development since it is not dependent on the measurement of code. The cyclomatic complexity measurement

provides an evaluation tool by which "goodness" of a module can be reviewed following its detailed design [Ref. 1 : p. 169].

D. SOFTWARE COSTING

The role of software in the military and private sector has grown considerably during the past decade. During the infancy of computing, software cost amounted to only a small percentage of the overall computer system. Today, software is the most significant portion of most computer systems. Accurate estimates of software development cost seldom occur, with the final costs normally running considerably higher. There are two fundamental problems which make accurate estimates of software development costs most difficult [Ref. 13 : p. 45]. These are:

- the high risks and uncertainties involved in software development
- the lack of a quantitative data base for previous cost estimates and final costs. i.e., "lessons learned"

In spite of these significant problems, cost estimate are made and will continue to be made with varying degrees of accuracy

This section will describe three current methods of cost estimating and provide a table for their comparison based on application [Ref. 13 : p. 15 - 17].

1. Analogy

This method estimates the costs for a new system based upon the the costs of a similar system. The cost estimate is adjusted to compensate for any differences between the two systems being compared. The analogy method is fairly simple provide accurate cost data for the existing system is available and the development methods and resources are similar.

2. Decomposition

As the method name suggests, a system is broken down into components and subcomponents until the level of decomposition makes it possible to estimate the costs fairly accurately. One approach of decomposition uses the analogy method previously described. In this approach, the process of decomposition is effectuated until the resulting level of decomposition can be compared with a similar component which already exists. A second approach of decomposition divides the system into components for which a level of effort can intuitively be estimated for each kind of activity that is needed to produce that component. This latter type of decomposition normally depends heavily upon the technical knowledge and experience of the estimator. The preassumption that underlies this method of cost estimating is that the costs for small systems, or components, can be accurately made. The total system is perceived as the aggregate total of its subsystem.

3. Parametric Models

As with the analogy method, the parametric model approach to cost estimating is also heavily dependent on the accumulation of past and accurate cost data for software development. Analyses of cost data permits the identification of cost variables and a quantification of their relationship to cost. Any new cost estimate can be derived by estimating the assigned values of the cost variables. Once this is done, the cost can be calculated using the equations which express the cost estimating relationships. The advantages of this method is that it allows for a rapid determination of cost estimates, using parameters whose values can be easily modified.

Table 3 [Ref. 13 : p. 16] provides a comparison of the three cost estimating methods discussed. Combinations of two, or more, methods may be used together, or separately to test the validity of an estimate. Resultant differences are adjusted to arrive at a reasonable estimate

TABLE 3
Comparison of Cost Estimating Methods

<u>TYPE</u>	<u>DESCRIPTION</u>
Analog	Compare to prior system
Decomposition	Divide into parts, activities
Parametric Models	Equations based on prior data about cost relationships
<u>TYPE</u>	<u>GOOD FOR</u>
Analog	. similar systems with similar resources, development process
Decomposition	. resource allocation . unique systems
Parametric Models	. rapid estimation . estimator's inexperience in software development . estimating risk
<u>TYPE</u>	<u>BAD FOR</u>
Analog	. unique systems . different environments
Decomposition	. initial estimates with no design . rapid estimation
Parametric Models	. systems different from data sample . poorly correlated data base

Automated costing systems provide another option for estimating. In these automated systems, the characteristics of the development organization, such as the staff's experience level, and characteristics of the software to be developed are described. Cost estimates are derived from this input data. As with the other three manual methods, the derived cost data will only be as good as the empirical data upon which it is founded. If no historical data exists, then the validity of the cost estimates is, indeed, questionable.

E. CHAPTER SUMMARY

McCabe's and Halstead's software metrics remain a controversial topic. But they do represent a revolutionary approach toward providing software managers with quantitative functions for estimating many heretofore elusive characteristics of software. The validity of Halstead's experiments have yet to be significantly tested. For those tests that have been performed, the size of the programs were generally small, and the subjects were college students vice professional software developers.

As compared to Halstead's metrics, McCabe's complexity measure can be applied during the earlier stages of software development since it is not dependent on the measurement of code. The cyclomatic complexity measurement provides an evaluation tool by which "goodness" of a module can be reviewed following its detailed design. Despite the criticisms that normally abound the proposition of new theories, both Halstead's and McCabe's metrics represent a giant leap toward adding quantitative measurements to a discipline that has defied them.

The military's justified and growing concern over frequent cost overruns for software development is forcing

changes in both the management and development of software. As such, new requirements and changes can be expected that will provide a more uniform and better control over cost and software development. This chapter has addressed three of the most common software cost estimating methods. The STARS program, Chapter VII, addresses a DoD-sponsored software initiative which will significantly alter and guide future software development efforts for DoD weapon systems.

IV. QUALITY SOFTWARE

"Software correctness remains the most elusive goal of computer science. As a result, software is the most unsafe, the least understood, and the most expensive component of total computer system cost. In contrast, cost of computer circuitry have shown a dramatic decrease, especially in the past 15 years, and computer hardware capability has improved." [Ref. 14 : p. 16]

A. BACKGROUND

The preceeding quotation was taken from an article authored by the Deputy Under Secretary of Defense (Research and Advanced Technology), Dr. Ruth Davis. It expresses the concerns shared by many DoD top officials relating to the both cost and safety risks associated with the development of today's computer systems.

As a percentage of total computer system cost, it is generally known that the cost of hardware has decreased dramatically over the the past 15 years while the cost of software has steadily increased. Today, software represents approximately 90% of the total computer system [Ref. 14 : p. 18]. There are two basic reasons to explain the change in the cost ratio between hardware and software. These are: [Ref. 15 : pp. 55 - 56]

- (1) Size: Today's software programs are an order of magnitude larger than they were two decades ago. This can be attributed, in part, to the increase in size (and simultaneous decrease in the cost) of onboard memory. An adaptation of parkinson's law suggests that program instructions will continue to increase until the limits (and frequently beyond) the available core is fully utilized.
- (2) Complexity: As nature of the applications being automated today are considerably more sophisticated than those applications of yesteryear. Both military and commercial survival strategies are becoming increasingly dependent on maintaining the competitive edge in computer superiority.

Software has become a primary vehicles for solving many of the new and changing problems facing the military. In many cases, changes to software is often viewed as an efficient and expedient way to solve a variety of emerging problems or threats facing DoD without having to change the existing hardware. Yet the virtues of software are often outweighed by its associated problems as described in chapter II.

It is not suprising that DoD has identified software as the most significant factor in determining the total cost of computer-based systems over their life cycles. Numerous studies have been conducted which show software quality as one of the most significant factors determining the life cycle cost of software. This chapter will present many of the characteristics of software quality and the means to achieve them.

B. DEFINING SOFTWARE QUALITY

Defining quality software is, in itself, a task. There are as many definitions of what makes software "good" as there are authors that write about it. Yet these definitions are not mutually exclusive. Each author has his own ideas of what the principle characteristics of quality software are, and each is right. Defining quality software is as difficult as defining the virtues of mankind. Air Force Regulation (AFR) 74-1, the "Air Force Quality Assurance Program," broadly and sensibly defines quality as "The composite of material attributes including performance." Other, more specific, definitions advocated by many of the "gurus" of software engineering will now be discussed.

Pressman [Ref. 1 : p. 148] suggests that good software has three essential qualities:

- the software works according to the specified requirements-- being as fast, efficient, and as functional as needed.

- the software is maintainable--it can be diagnosed and modified without great difficulty.
- the software is more than merely lines of code--it includes all the supporting documents to ensure that the first two qualities are achievable.

According to Pressman, good software is based upon good design, and good design can be gauged by applying a number of software engineering measures and heuristics.

DeMarco [Ref. 8 : pp. 198 - 200] prefers to define software quality as "the absence of spoilage" [Ref. 8 : p. 200], with the term "spoilage" meaning the amount of effort required to find and remove faults introduced during the software development process. Equating this amount of effort to its commensurate cost, Demarco provides a formula to quantify software quality: [Ref. 8 : p. 200]

$$\text{Quality} = \frac{\text{Summation of Defect Diagnosis and Correction Cost}}{\text{Program Volume}}$$

In which Program Volume is measured per thousand lines of executable code (KLOEC)

C. CHARACTERISTIC OF SOFTWARE QUALITY

One of the most comprehensive and significant works written to provide a framework for assessing the issues associated with software quality is found in the study conducted by Boehm, et al., titled Characteristics of Quality Software [Ref. 16]. This section will present many of the highlights reported by this study.

In developing a methodology for the assessing the quality of software products, the authors concluded that "calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth." [Ref. 16 : p. 3-2] A major problem in developing a single metric for gauging the quality of software is that many of the characteristics of software are in conflict with

one and another. For example, requiring a high degree of software portability is achieved at the expense of software efficiency. Code conciseness is at odds with maintainability, understandability, and so forth. As such, the study developed a relational set of important software characteristics which were reasonably exhaustive and non-overlapping. This set of characteristics would serve to define a working context for collection and formulation of a set of candidate metrics used to assess the degree to which the software possessed the respective characteristic. Figure 4.1 shows the resulting characteristic set and their hierarchical interrelationships [Ref. 16 : p. 3-19]. Definitions for each of the represented characteristics is provided in

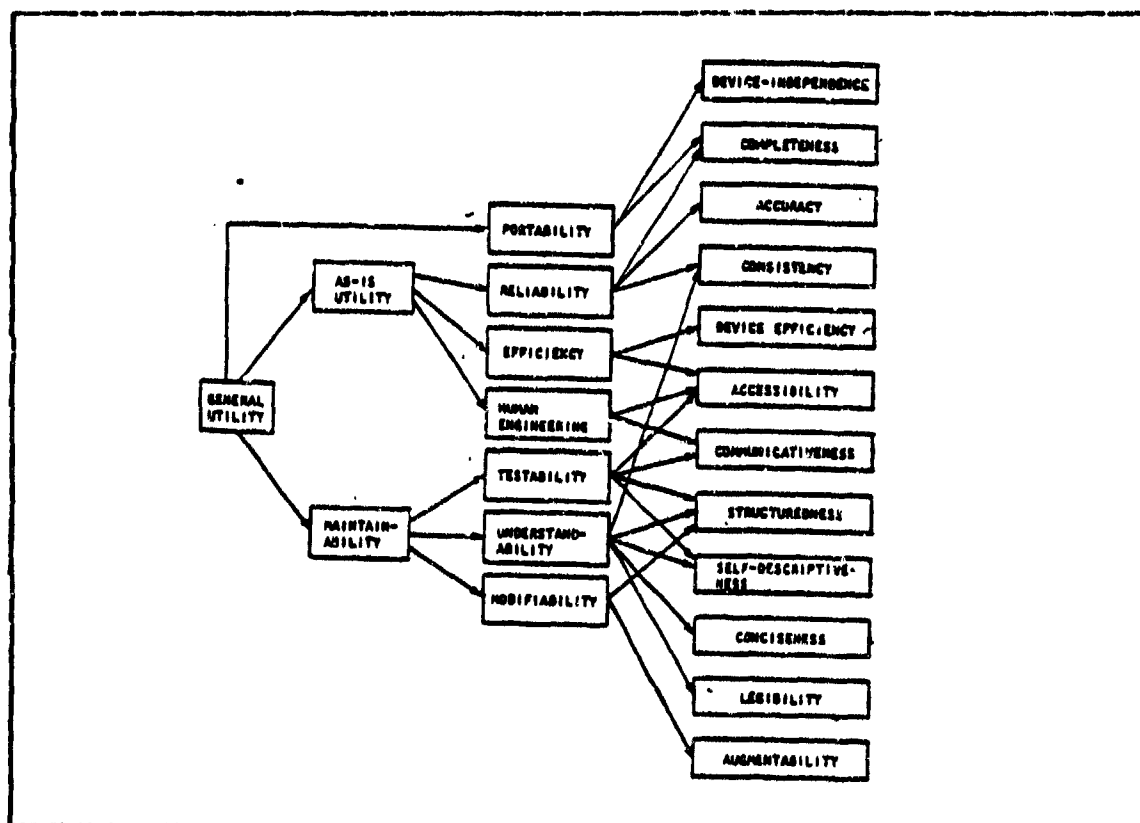


Figure 4.1 Characteristics Tree

appendix A. The characteristics depicted in Figure 4.1 are categorized in three hierarchical levels. The higher-level structure is oriented toward accommodating various user needs and priorities for a software product. For example, "As-is" utility is analogous to the "black box" understanding of a system; the user is concerned with only the inputs and outputs of the product and need not understand the its internal code, nor how to modify or test it. If the product is going to be changed by the user, then maintainability requires that the user be able to understand, modify, and test the product.

The lower-level structure depicts those primitive characteristics, which, although strongly differentiated from each other, "combine into sets of necessary and sufficient conditions" [Ref. 16 : p. 3-25] to define the intermediate-level characteristics. The primitive characteristic provide the foundation for formulating the metrics used to quantifiably measure a software products relative possession of those characteristics described in both the high-- and intermediate layers.

D. QUALITY ASSURANCE

The preceeding section described many of the attributes associated with good software, as well as their interrelationships. The purpose of this section is to offer a framework through which quality software can be achieved through planning, specification, and monitoring of quality assurance (QA) activities.

The purpose of software quality assurance, in short, is to assure the ultimate quality of the delivered software. A formal definition of quality assurance is provided by AFR 74-1, which defines it as:

" A planned and systematic pattern of all actions necessary to provide adequate confidence that material, data, supplies, and services conform to established technical requirements and achieve satisfactory performance."

Another definition for quality assurance is offered by Pfau [Ref. 17 : p. 2] who also helped remove some of the subjectivity that surrounds the term "quality" by stating:

"Quality assurance is the name given to the activities performed in conjunction with a software product to guarantee the product meets the specified standards. These activities reduce doubt and risks about the performance of the product in the target environment."

Both of the above definitions are reflective of the direction that QA has taken over the past two decades toward a total life-cycle perspective. This evolution of QA has been divided into three separate generations [Ref. 18 : pp. 2 - 4]. It is important to understand the differences in these generations in order to avoid the serious pitfalls implicitly and explicitly expressed in the first two.

First Generation--Test-Oriented QA: This QA generation basically equated QA to software test programs. Tests plans and procedures, types of test, and methods of formal verification of performance/design requirements were all essential to the testing activity.

The obvious and major pitfall of the test-oriented QA generation is that "you don't test quality into a software product." [Ref. 18 : p. 2] Even though testing facilitated the discovery of deficiencies, the discovery normally took place too late in the development process to allow their relatively inexpensive resolutions.

Second Generation--Development-Oriented QA: Due to the inherent failure mechanisms built into test-oriented QA, corrective actions were taken by an attempt to make the

developing contractor responsible for the quality shortcomings of the products they produced. This was done by assuring that the software delivered under contract fully complied with the requirements of the contract.

The pitfall to this QA approach is as limited as the contracting officer technical knowledgeable in the broad discipline of software engineering. Contract delivered what was specified, nothing more.

Third Generation--Life-Cycle-Oriented QA: In this generation, QA is built into the software from "day one." The effort is properly focused on the early definition phases for planning and specifying contractual provisions concerning software attributes. Figure 4.2, [Ref. 1 : p. 25] illustrates the cost impact of introducing changes during

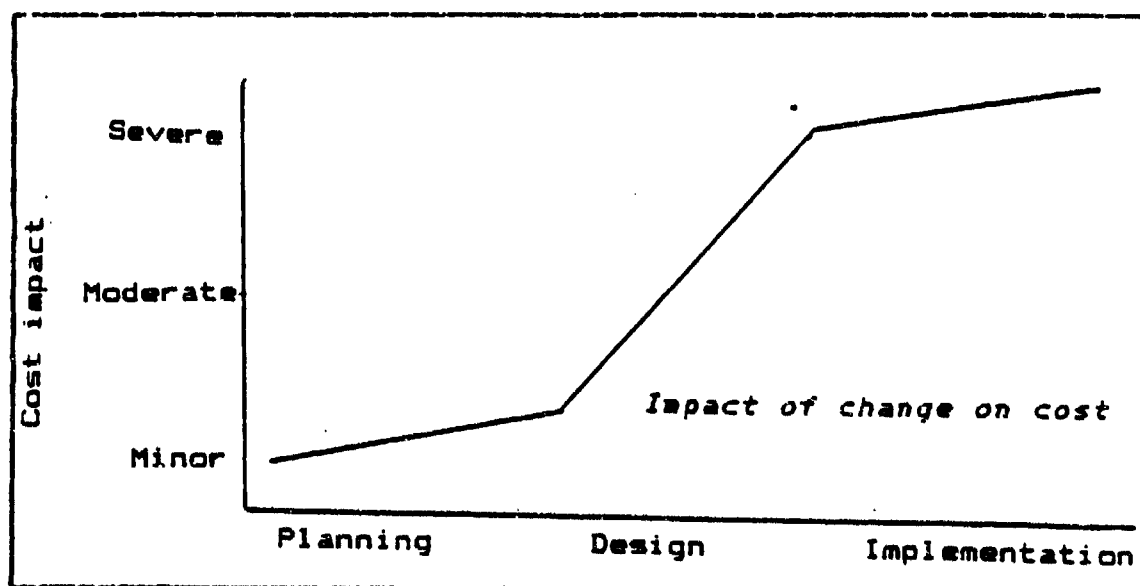


Figure 4.2 Cost Impact of Changes

various phases of the software life-cycle. Emphasis is placed on the clear definition of those software characteristics that were discussed in the first section of this

chapter, such as maintainability and portability, which have a significant affect on the quality of the product over the system's life-cycle. The importance of the life-cycle-oriented QA approach and its impact on life-cycle costs following software development and implementation is discussed at length in the chapter on software maintenance.

E. IMPLEMENTATION OF A SOFTWARE QUALITY ASSURANCE PROGRAM

The preceeding section addressed the definition of software quality assurance, as well as its evolution to the present life-cycle-oriented perspective which recognizes that to achieve the highest quality of software it is necessary to include quality checks throughout all phases of the software life-cycle.

This section will discuss the military standards for the implementation of software quality assurance (SQA) programs in defense contracts. The successful implementation of these programs will provide early visibility and managerial controls to detect, report, analyze, and correct software deficiencies. Although the focus of this discussion will be on defense contracts, the methods addressed herein may be equally beneficial to in-house development efforts.

The two most significant military standards affecting the establishment of SQA programs for defense contract are: (1) MIL-STD-52779(A), "Software Quality Assurance Program Requirements," providing the basic elements required in an acceptable SQA program, as well as customer evaluation criteria, and (2) MIL-STD-1679, "Weapon System Software Development," providing detailed software development standards for the entire weapon system software development process. Both the software manager and procuring agent should be familiar with their contents since, together, these standards provide an effective means to evaluate any software development program [Ref. 19 : p. 108].

In their article [Ref. 19], Dobbins and Buck discuss five areas of control which follow the typical chronology of software development. These are: (1) procuring agency evaluation, (2) design inspection, (3) code inspection, (4) test, and (5) library controls. The remainder of this section will address each of areas separately.

1. Procuring Agency Evaluation

From both a cost and effectiveness standpoint, the consequences are too important to accept at face value the claims that a strong SQA program exists in their organization. There must exist some means to evaluate the potential contractor. Major quality items must be addressed as early as possible in the planning process prior to the Request for Proposal (RFP) preparation. These quality items should include those attributes considered as an integral part of the software design, development, test and evaluation, and maintainability issues. Table 4 [Ref. 18 : p. 33] provides a number of factors with which to evaluate bidders' responses to the RFP process.

Often the program manager and procurement agency will have insufficient experience and technical background to properly identify essential QA issues needed for inclusion in the RFP nor the means or time to evaluate the contractor proposals. In these situations there are alternatives resources available to evaluate the contractor. The first of these is the Defense Contract Administrative Service (DCAS). The program manager can hire the services of software engineers acquainted current military QA standards. Depending on the end-application of the software product, there are other government organizations through which assistance can be sought. Other alternatives include hiring the services of a commercial contractor or consulting firm. Regardless of the resource used, a sound means for contractor evaluation and selection is essential.

TABLE 4
Evaluation Factors in Bidder Responses

FACTORS:	BIDDER EVALUATION CHECKLIST:
Completeness	Does bidder's response cover all area as requested in the RFP?
Scope	Is the scope of the bidder's response consonant with the project's objectives and the level of detail in the RFP ?
Compliance Documents	Are all compliance documents regarding the software design identified?
Design Review Board	Does the bidder propose to have design Review boards?
Event Sequences	Are the reviews of software design done in proper sequence?
Problem Reporting	Does the bidder propose to formally identify all design problems?
Action Item Follow-Up	Does bidder provide assurance for effective follow-up of all action items resulting from reviews?
Past Experiences/Resources	What experience does the bidder have with QA? Does he have a good resource base?

2. Design Inspection

Although MIL-STD-1679 specifies that "walkthroughs" should be used as the means to collect statistics during the design phase of software development, the process has evolved to the more formal procedure termed "design inspection." The difference between the two approaches is one "of

rigor, not intent" [Ref. 19 : p. 110]. The walkthroughs were informal examinations of the software product by its authors' technical peers. Little documentation was kept, and no training requirement placed on the participants of the walkthrough.

As with the walkthrough, the design review is a peer inspection process performed by teams that inspect one another's work. Unlike the walkthrough, the design review is a formal process in which records are kept, and participants undergo considerable training requirements. It is conducted when the design is completed, prior to the actual coding effort. The inspection team is led by a moderator. The ideal moderator is not only trained in the technical aspects of software engineering, but in the psychological^{*} aspects of software development.

The moderator promulgates required inspection material to the team in advance of the inspection. Each team member reviews the material and records comments before the inspection meeting. During the meeting, discussion is reserved for major error, i.e., those errors that will prevent the program from functioning properly. Minor errors simply recorded for subsequent correction. If more than 5% of the program design must be changed due the errors, the entire design will be reinspected. Otherwise, the moderator will assure that the errors discovered during the design inspection are corrected before proceeding into the next phase.

^{*}For an more in depth discussion of the psychological aspects involved in software development, the reader is referred to Gerald Weinberg's book The Psychology of Computer Programming.

3. Code Inspection

Programming coding may begin only after successful completion of the design inspection. MIL-STD-1679 requires top-down structured programming and identifies the specific code constructs allowed. Figure 4.3 [Ref. 20 : p. 62], illustrates the five basic code structures, each having a single

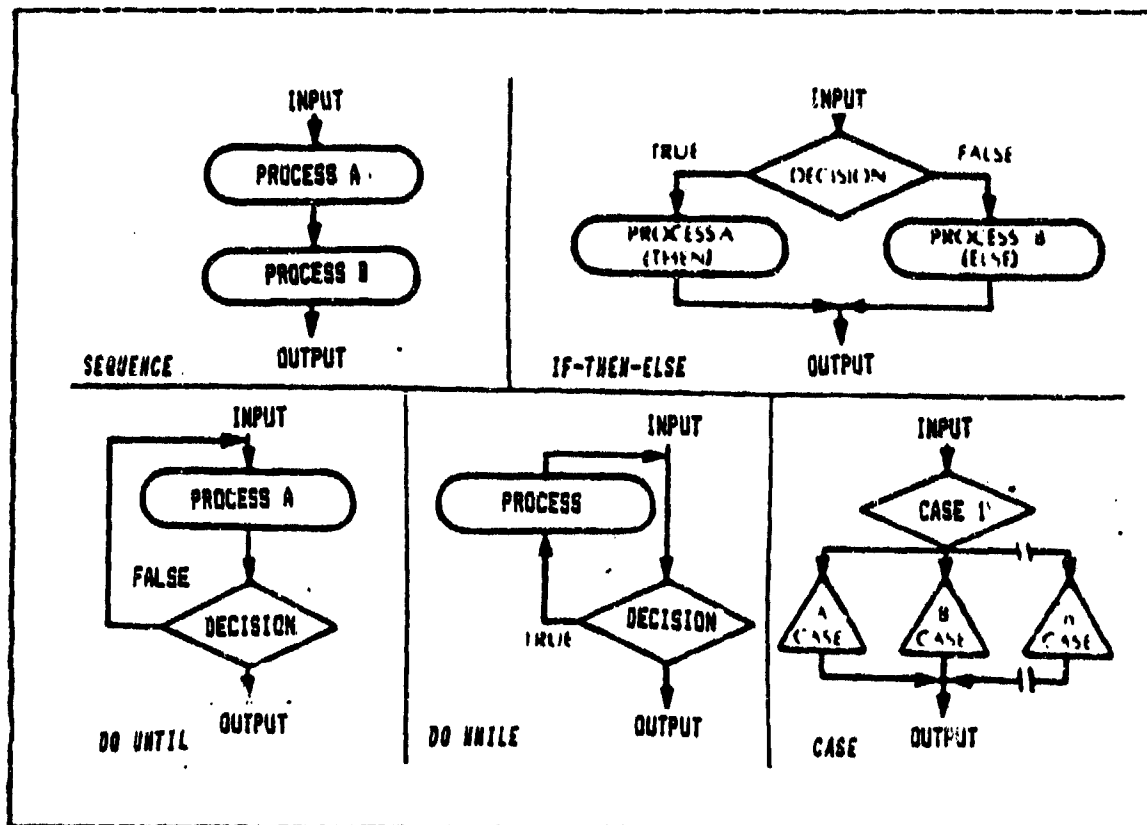


Figure 4.3 Basic Code Structures

data entry and exit.

Once the program is coded and successfully compiled, it is inspected. The process for inspecting code is nearly identical to that discussed for design. Not only is the code inspection a method of discovering coding errors, but just as importantly it assures that the code adheres to the approved design.

4. Test

Software testing accounts for the majority of technical efforts expended in software development. Its objectives are to uncover software errors, and to provide assurances that the software performs its technical and operational requirements.

An effective SQA program must start at the front end of software development, with the requirements specified in the RFP, addressing the totality of the testing to be performed. Three measures of software testing relating to the RFP include: [Ref. A059H068 : p. 19]

- (1) The analysis of software requirements for testability.
- (2) The identification of the contractor's software testing activities as part of his Software QA Program.
- (3) The review of test documentation and certification of test results.

Testing requirements specified in MIL-STD-1679 require that the system software do more than just meet the specifications. Software must also be subjected to a third-party⁵ "stress test," in which the program is judged unsatisfactory if the program execution can be stopped for whatever reason. To achieve a degree of software quality sufficient enough to pass this type of testing, it is vitally essential that the software development program incorporate programs of error detection and prevention well in advance of the actual testing period.

⁵As defined in MIL-STD-1679, the third party is neither the contractor nor the procurement agency.

5. Library Controls

A key element in any SQA program is the software library which provides visibility and control of the products documentation and programs. Among the mandatory controls stipulated by MIL-S-52779(A) is the control to prevent unauthorized access. Other essential activities in a software library are the documentation and program storage, and retrieval and change processing. MIL-STD-1679 requires a Software Change Control Board (SCCB), which must authorize any changes to the controlled library.

F. PARTING COMMENTS

The underlying goal of software development is to deliver quality software. In doing so, it is vital to examine the characteristics of quality, and their interrelationship, within the context of the user needs and ultimate application of the program. To understand the characteristics of quality software is to understand the founding principles of software engineering. To produce quality software is much more. The implementation of a software quality assurance program is the vehicle through which these principles are applied and the goal of software development realized.

The benefits derived from quality software support the saying that "quality is free." But more importantly, as will be addressed in the next chapter, future cost-avoidance during the maintenance phase leaves no practical alternative to acceptance of only quality software.

V. SOFTWARE MAINTENANCE

This chapter deals with the last phase of the software life cycle. Canning [Ref. 21 : p. 2] appropriately categorized software maintenance as an "iceberg," initially revealing only a small portion of maintenance requirements, but hiding an enormous potential for future problems and costs under the surface. With few exceptions, computer programs are always changing in order to correct latent errors, add enhancements, and seek performance optimization of the software. A succinct definition for maintenance can be given as "that activity which is concerned with making changes to software for the purpose of improving or correcting the software." [Ref. 22 : p. 2] Maintainability is defined 22 as "a property of software which makes the maintenance activity easy to perform, i.e., changes to the software are easy to incorporate and do not lead to new errors in the software." This chapter will primarily address issues of maintainability which by necessity must be considered during all phase of the software life cycle.

A. CATEGORIZATION OF MAINTENANCE ACTIVITIES

Maintenance is much more than just fixing errors that escaped detection during the pre-delivery tests and evaluations. Maintenance has been categorized [Ref. 1 : p. 323] into four activities that take place after the program is released for use. These are corrective maintenance, adaptive maintenance, perfective maintenance, and preventive maintenance. Each will now be described.

Corrective maintenance is the process that includes the diagnosis and correction of latent errors that avoided

detection prior to the implementation of the program. It is impractical, if not impossible, to exhaustively test complex programs in order to guarantee 100% error-free software.

Adaptive maintenance are those modifications made to the program as a result of changes to the environment in which the program must operate. As an example, it is often quicker and less expensive to modify software rather than the hardware in order to modify weapon systems to satisfy new threat situations.

Perfective maintenance is the process used to accommodate recommendations for new capabilities, changes, and general enhancements requested by the user of system programmer.

Preventive maintenance takes place when software is changed in order to improve its future maintainability. This type of maintenance remains a rare practice in software engineering.

Based upon a study of 487 software development organizations by Lientz and Swanson, as summarized in reference 1, 50% of all maintenance is perfective. Corrective-- and adaptive maintenance account for 21% and 25%, respectively. All other types of maintenance account for only 4%.

B. TANGIBLE MAINTENANCE COST

Although considered by many software engineers as the less glamorous and unexciting phase of software development, maintenance accounts for the majority of the dollars spent throughout the software life cycle. The cost of maintenance has shown a dramatic and steady increase over the past two decades. As depicted in figure 5.1, one author [Ref. 1 : p. 326] estimates that maintenance cost as a percentage of the total software budget will have grown from 35-40% in 1970 to 70-80% in 1990.

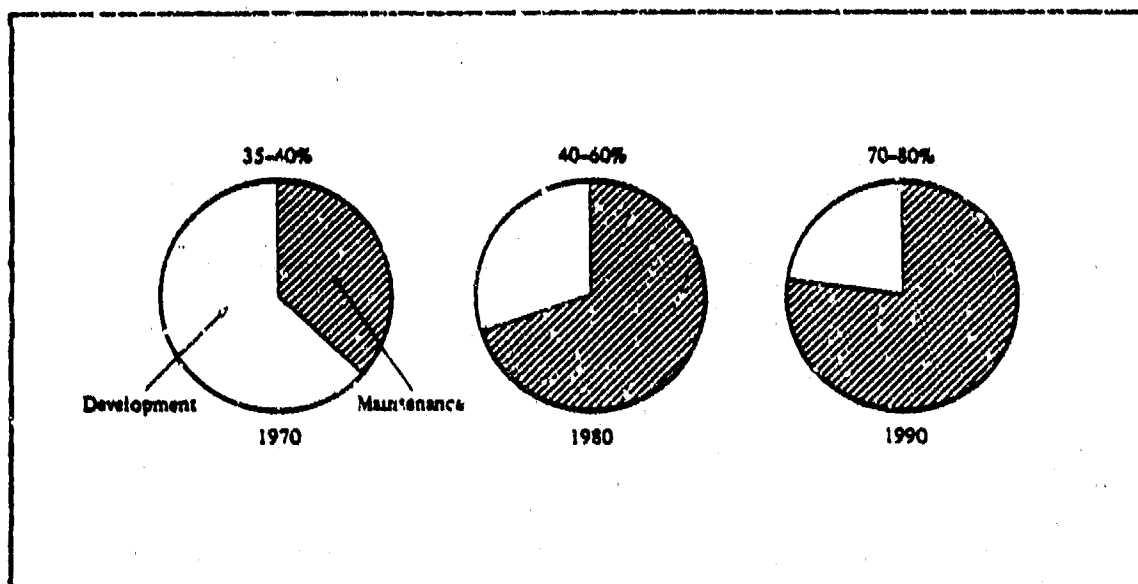


Figure 5.1 Maintenance Cost as Percentage of Budget

Although empirical data is available to account for total software life-cycle cost allocatable to maintenance, maintenance costs are very difficult to estimate in advance. It is known, however, that maintenance costs are often dramatically underestimated by both industry and government during the pre-deployment phases of system acquisition. To illustrate this point, Boehm [Ref. 23 : p.127] estimated that it took \$30 to develop a line of code (LOC), but the cost per LOC skyrocketed to \$4000 in the maintenance phase. Although \$4000 per LOC may seem unreasonably high, it is not unusual to incur such high costs for maintaining mission-critical software in DoD weapon systems.

Although there is not a set of universal factors that can be applied to all software development projects to accurately estimate the relative cost of program modification in each of its life cycle phases, figure 5.2 illustrates the exponential rise in maintenance costs in each of the phases. [Ref. 24 : p.14]

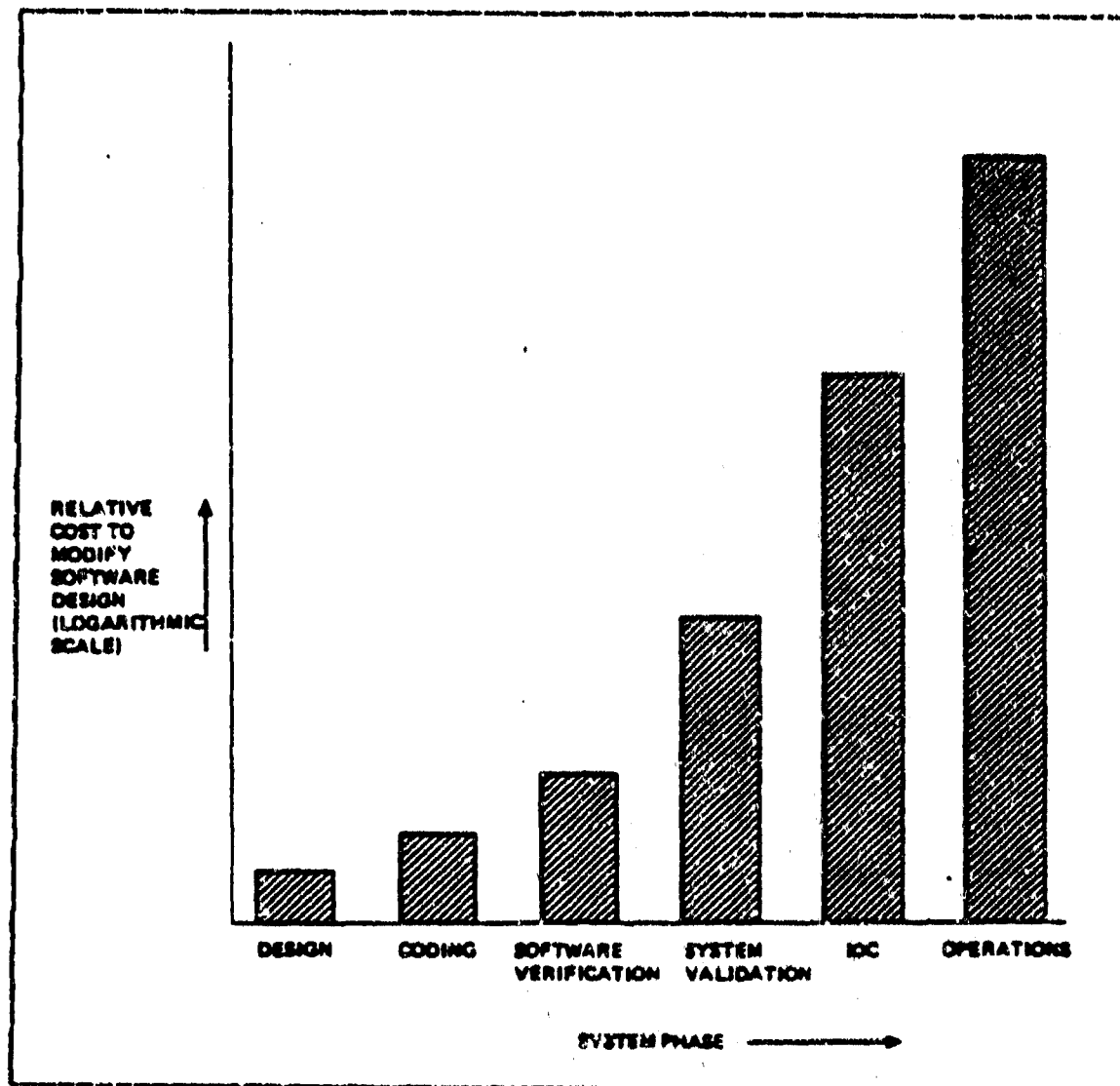


Figure 5.2 Life Cycle Maintenance Costs

It is apparent that there is more potential for realizing life cycle cost savings by devoting more planning during the earlier phases in order to minimize the requirement for maintenance during subsequent phases. One of the primary reasons for the high cost during the later phases is due to the "domino effect" of changes that must be promulgated throughout the entire system for what may seem at first to be a simple code modification requirement.

C. VARIABLES AFFECTING MAINTENANCE COSTS

As mentioned in the preceding section, an accurate estimate of maintenance costs for a particular program is very difficult. Sommerville [Ref. 25 : pp. 198 -199] has identified five relatively unpredictable factors that contribute to the difficulties involved in deriving cost estimates for maintenance. These factors include:

- (1) The application being supported. The better the application being supported by software is understood, the better the system requirements can be stated. The more definitive the system requirements are, the less perfective maintenance will be required in the future.
- (2) Lifetime of the program. Toward the end of the program life, a deterioration of the program structure occurs due to the multitude of modifications that the typical program has gone through. Historical evidence suggests that program lifetime is traditionally much longer than originally estimated. Many systems today are still running on programs that were coded in the early 1960's.
- (3) Dependence of the program on its external environment. The closer a program is tied to factors in the external environment, the more flexible and expandable that program must be to accommodate modifications due to changes in environment in which it operates.
- (4) Staff stability. It is normally easier for the original author of the program to make changes than for another programmer who must gain an understanding of the program by studying its documentation. Pressman [Ref. 1] uses the term "alien code" to describe those programs that are extremely difficult to understand by those that must maintain them. Reasons for alien code include: (1) no current member on the maintenance staff was involved in the development of the program, (2) poor design and documentation of the program, and (3) modularity and structure design concepts were not used in the development of the program. High turnover within the programming profession has made it a rare occurrence for the same individuals to develop and maintain a program throughout its life cycle.
- (5) Hardware stability. Software is designed to be compatible with the hardware that will support it. Changes to the hardware configuration will likely result in requirements for software modification.

D. INTANGIBLE MAINTENANCE COSTS

The direct cost of maintenance, although considerable, may be of secondary concern when compared with the less obvious and intangible cost of maintenance. A quote by Daniel McCracken [Ref. 1] summarizes one of these intangible costs, the development opportunities that are lost due to the resources that must be allocated to maintenance efforts:

"Backlogs of new applications and major changes that measure in years are getting longer. As an industry, we can't keep up--let alone catch up--with what our users want us to do."

McCracken alludes to what Pressman ¹ call a "maintenance-bound" software development organization which is no longer capable of producing new software because all its resources are devoted to the maintenance of existing software. Pressman lists other intangible costs including:

- ~~Customer dissatisfaction~~ due to the untimely response by the software development organization to the user's development and maintenance requests
- ~~Reduction of software quality~~ brought about by latent errors introduced during the maintenance of software
- ~~Upheaval of development efforts~~ as personnel and other resources are "pulled" to work on maintenance tasks

E. BUILDING MAINTAINABLE SOFTWARE

Economic and efficient support of software is best achieved when its maintainability is integrated into the total development effort from day one. The maintainability of software can be quantitatively measured based on the ease by which it can be understood and changed [Ref. 26 : p. 14].

Software understandability is a function of the design and documentation. It is easy to understand due to its logical and simple structures, and it is supported with

documentation that permits an examination of the implementation without losing an understanding for the entire picture. Software changeability, on the other hand, is a function of the design and implementation. As an example, implementation of modular independence facilitates changes to a selected segment by minimizing the degenerative affect on other segments.

1. Structured Methodology

Building maintainable software is based on the usage of a set of software engineering tools and techniques that together form a structured methodology for software development. As the authors [Ref. 26] write, the principal elements of this structured methodology include:

"Structured Analysis. A process for developing the functional, data, and interface requirements of the software design by constructing a logical model of the system process.

Structured Design. The process of subdividing the software into hierarchical modules in a way that tends to minimize module independence.

Structured Programming. The discipline of implementing the control structure of software modules using a restrictive set of structures.

Program Design Language (PDL). Language processors that are used to document software designs in a structured top-down manner."

Although there are many other disciplines and concepts that must also be considered as part of a complete structured methodology, such as top-down implementation, structured walk-throughs, chief programmers teams, and so forth, these managerial disciplines are used primarily for the software development effort. The methodologies underlined above will have a visible affect on the software product long after its development is completed. A detailed description of the characteristics of each of these elements is provided in Appendix C.

Although there are many other disciplines and concepts that must also be considered as part of a complete structured methodology, such as top-down implementation, structured walkthroughs, chief programmers teams, and so forth, these managerial disciplines are used primarily for the software development effort. The methodologies underlined above will have a visible affect on the software product long after its development is completed. A detailed description of the characteristics, as provided in [Ref. 26] of each of these elements is provided in Appendix B. A more general discussion of each will now be presented.

2. Structured Analysis

Structured analysis is often considered the starting point in the set of structured design techniques. The main objective of structured analysis is to build a logical model of the desired system. This should be done to the greatest extent possible without premature consideration of physical implementation.

In its simplest form, the logical model is a pictorial representation with accompanying narration describing the functions, and their interrelationships, of the system that they comprise. Examples of the some of the most popular forms of these graphical representations include process and information flowcharts, data flow diagrams, hierarchy chart plus input- processing-output chart (HIPO charts), and procedure analysis charts.

The net result of the this structured analysis process should be a logical model that defines the complete system which reflects all facets of the system specification and software requirement document. The model should be a form of communication easily understood by both technical and nontechnical personnel, alike. Through the use of this model, the system analyst should be to develop system

requirements without undue consideration of physical implementation constraints. On the other hand, nontechnical users should readily be able understand how the required functions fit together within the context of the whole system.

3. Structured Design

Structured design is the process of decomposing the software design into hierarchical modules in a manner that leads toward independence of modules. Benefits of structured design to the development and maintenance of software include increased understandability of the system and a minimization of the cost inherent in modification.

Modularity is the key element of structured design. It allows for software to be better managed. Large monolithic (i.e., single module) programs are often unintelligible to the reader. Modularity is based on a "divide and conquer" concept, breaking complex problems into comprehensible and manageable components. Two primary measurements of modularity are (1) cohesion, and (2) coupling.

-- Cohesion is the "relative functional strength" [Ref. 1: p. 158] of a module. A module is said to be cohesive if it performs a single task within a program, requiring little interaction with other program code external to its boundaries. In general, design should attempt to realize the highest degree of module cohesion.

-- Coupling is a measurement of the connectivity among other modules. It is based on (1) the interface complexity between modules, (2) the place at which entry and reference are made to a module, and (3) the type of data that passes across the interface [Ref. 1: pp. 161 - 162]. The designer should strive for the lowest degree of module coupling.

Clearly, then, the objective of structured design is to minimize the relationships between modules through the maximization of the functional strength of each.

4. Structured Programming

Structured programming is the discipline of implementing module functionality through the use of a limited set of programming structures.

Structure programming uses top-down design by starting with the top-level module and decomposing it into lower-level modules that it will call upon. This decomposition process repeated as often as necessary until the bottom-level modules are defined. At this bottom level, modules make use of built in operators and functions; they do not call on any other module. Each module is separately coded using the basic set of program instructions. An objective of structured programming is to make the design match the structure of the program.

Any program, regardless of its size and complexity, can be designed using three basic programming structures. The use of this set of programming structures reduces the procedural design of the program to a small number of predictable operations, greatly facilitating the development and maintenance of software. These structures are illustrated in Figure 4.3.

5. Program Design Language

Program Design Languages (PDL) are language (text) processors that are used to document software designs in a structured top-down fashion. The goal of a PDL is to replace or support traditional forms of documentation of program design.

The primary benefits of a PDL are: (1) the documentation that it produces is normally easier to read and understand than flow charts, and (2) the documentation is always easier to change than are flow charts. Both of these advantages are essential during maintenance activities.

F. PARTING COMMENTS

The maintainability of software is inseparable from the degree of quality that was built in prior to the maintenance phase. Sound software engineering practices, coupled with the implementation of managerial controls in maintenance activities, offer the key to improved productivity and the reduction costs associated with maintenance activities.

VI. DOD STANDARDIZATION AND SPECIFICATIONS

A. INTRODUCTION

In 1980, it was estimated that DoD spends about \$7 billion a year on software [Ref. 28 : p. 3]. This amount has been steadily increasing as DoD becomes increasingly dependent on larger and more complex software products to support this generation of sophisticated weapon systems. The upward-spiralling trend in the cost of DoD software has naturally become an area of great concern to officials in both military and government. This concern has led to a number of management initiatives in DoD, several of which will be discussed in this chapter. At the heart of these initiatives is the standardization of computer technology and software. Standardization is seen a means for reducing costs associated with the development, operations, and support of DoD computer systems.

B. SPECIFIC INITIATIVES

In her article [Ref. 29 : pp. 37 - 47] Becker describes three distinct, but interrelated, initiatives that reflect the DoD standardization effort. These initiatives are as follows:

- (1) The Army's development of a Military Computer Family (MCF)
- (2) The adoption of Ada as a higher order language (HOL) for development of embedded computer software.

⁶An instruction set architecture can be described as the rules and procedures by which hardware executes instructions or computer software. It can also be defined as the structure of a computer that a programmer must know to write time-independent machine language [Ref. 29 : p. 39].

- (3) A proposed DoD instruction set architecture⁶ (ISA) standard (Draft DoD Instruction 5000.5X).

The first two of these initiatives will be summarized from Becker's article. In addition, this section will address standardization efforts by the Joint Logistic Commander's (JLC's) panel of Computer Resource Management (CRM).

1. Military Computer Family

The distinguishing characteristic in the Military Computer Family (MCF) initiative is a common instruction set architecture. The efforts to develop MCF began in the mid-1970's with an intensive review of the Army's mission-critical software. The Army first attempted to obtain an existing ISA through a licensing agreement from the commercial sector. Following an extensive evaluation of this first step, the Army concluded a licensing-agreement approach was severely limited for a number of reasons: [Ref. 29 : p. 41]

- (1) The adoption of a commercially-available ISA was perceived as placing unnecessary technical and administrative restrictions both on the participating vendor and the Army.
- (2) The protection and scope of a commercial ISA were perceived as a potential hindrance to the wide usage being considered by the Army.
- (3) Adopting a single firm's ISA was viewed being of unfair advantage to one company or a selected segment of the industry, thus greatly restricting competition.

As an alternative, the Army engaged the services of Carnegie Mellon University to develop an ISA, which became known as "Nebula" ISA and designated by MIL-STD-182A. Nebula has been rated as both an effective and advanced ISA. Under a memorandum of agreement, the Army and the Air Force have worked jointly to develop and control the Nebula program.

Using Nebula as the keystone, the Army has engaged in a multiphased competitive-procurement process to develop

a prototype computer model which will be at the heart of the MCF. Although a number of competing companies will be involved the the pre-production phases of this development effort, only one company will be selected to enter the production phase. The number of units acquired during the production phase will be based on unit cost as stipulated in a requirement agreement that was used as a criteria in the final competition.

Technological infusion is a major consideration of the MCF strategy, ensuring that the MCF has current technologies included in the mission-critical systems that are fielded. The Army hopes the the MCF program will result in improved survivability and logistics, as well as a reduction of life cycle costs of the MCF systems.

2. Ada

About the same time that the Army began its MCF program, the Department of Defense recognized the need for a state-of-the-art program for embedded computer applications. In the mid-1970's, DoD was spending about \$3 billion a year on software, with the greatest portion going for embedded systems [Ref. 30 : p. 268]. After concluding that the existing programming languages were inadequate for satisfying future software development needs, DoD set up the Higher-Order Language Working Group (HOLWG) to investigate the development of a new programming language. During the four year period, 1975 to 1979, HOLWG published a series of mandatory specifications for the new language. Each set of specifications were more detailed than the preceding set, as implied by their names: [Ref. 30 : p. 269]

In 1977, HOLWG studied 26 languages, none of which was able to meet the required specifications. A competitive language design effort was initiated 1977. By 1979, the 16

TABLE 5
Ada Specifications

Strawman	1975
Woodenman	1975
Tinman	1976
Ironman	1978
Steelman	1979

original propositions submitted by industry were reduced to one. The winning language was designed by CII-Honey-Bull, and was re-named "Ada."⁷ [Ref. 30 : p. 269]

The Ada Joint Program Office, under the Deputy Under Secretary of Defense (Research and Advanced Technology), is responsible for the management and implementation of all Ada-related activities.

Ada is not without problems and limitations. Designed to facilitate a wide range of applications, Ada is an extremely complex and large language. Using context-free grammar tokens as a measurement, Ada is estimated at 1600 tokens long, Pascal at 500, and Algol-60 at 600. The development of Ada has already been subjected to many of the same criticisms received by IBM during their effort to design FORTRAN VI. The resulting language, which incorporated features from FORTRAN, Algol, and COBOL was unrecognizable as FORTRAN and was subsequently renamed PL/I. PL/I represents the classic "Swiss Army knife" approach to software design in which all conceivable features that a user might need are built into a single language. The final product being too large and complicated for most programmers to

⁷Ada is a trademark of the Department of Defense, named for Augusta Ada Lovelace, the world's first programmer, and daughter of Lord Byron.

master [Ref. 30: p. 182]. As with PL/I, the size of Ada may lead to similar problems as well as inefficiencies in real-time application.

Provisions and exceptions will have to be made by the DoD for existing computer systems whose software is written in other languages besides Ada and where conversions to the Ada language may not always be possible or feasible. However, it will be expected that Ada will be applied where possible, and deviations to this requirement discouraged. Full implementation of Ada is bound to take some time since the language, itself, is still in a state of transition and because of the huge investment DoD presently has in programs written in other languages.

It typically takes the better part of a decade for a new language to become fully established, but Ada's initial acceptance by the commercial sector has been good. Convinced that the use of Ada will increase "flexibility and aid in the greater utility of its software packages," [Ref. 29 : p. 43] IBM has begun to implement a version of Ada. Another indication of the general acceptance of Ada is the fact that the Ada language is in its final stages for consideration by the American National Standards Institute.

3. Joint Logistics Commanders Workshop

In April, 1979 the Computer Software Management (CSM) subgroup of the Joint Logistic Commanders (JLC) Joint Policy Coordinating Group on Computer Resources Management (JPCG-CRM), sponsored a workshop at the Naval Postgraduate School in Monterey, California--appropriately entitled Monterey I. The purpose of the workshop was to review the services' software acquisition guidelines, management policies and procedures, and standardization efforts to see if there was a basis for the adoption of joint-service guidance in these areas. Monterey I concluded with the recommendation

that the services should adopt common software policies, development standards, and documentation standards instead of continuing with each of the service's unique and often-time redundant efforts pertinent to these areas. The advantages could be attributed to the adoption of joint-services standards: 1) economies, and 2) the best methods of each service could be adopted for use by all [Ref. 31 : p.192].

Other findings of the workshop included: [Ref. 32 : pp. 2-1 - 2-9]

- (1) No general policy exists for defining a common software acquisition framework for the joint services.
- (2) A number of diverse regulations and standards exist within DoD covering the various aspects of software acquisition and software documentation.
- (3) MIL-S-52779, "Software Quality Assurance Program Requirements," has been widely used since 1974, and has become an official joint services standard. The application of this standard has been met with varying degrees of success. Its application has been considered unacceptable due to the imposition of additional schedule and budget requirements. Furthermore, DoD plant representatives and DCASR personnel have found it most difficult to use in the evaluation and monitoring of software development contractors.
- (4) Lack of recognized software acceptance criteria, a lack of DoD standardization, and a lack of historical data upon which to base acceptance criteria and procedures.

Recommendations included the following:

- (1) Develop a general policy framework for the joint services to address the entire software life cycle.
- (2) Develop a unified set of acquisition and development standards for tri-service service application.
- (3) Develop a comprehensive set of data item descriptions (DID's), subsets which could be used for and software contract.
- (4) Generate a DID for contractor's software quality assurance plan as a joint service DID.
- (5) Define and develop software acceptance policy, procedures and criteria for the acquisition of defense system software.

The Monterey I workshop concluded with the CSM developing a plan of actions and milestones for the

implementation of the recommendations listed above, which were subsequently approved by the JLC's.

Since receiving the go-ahead from the JLC's, significant process has been made in carrying out the implementation plan [Ref. 33 : pp. 21 - 22]. The basis for this effort was centered around the definition of the software development life cycle, with the data item descriptions and standards integrated into the appropriate phases of the life cycle. Twenty-five basic DID's, defined for this purpose, replaced a total of over 200 previous ones. This has significantly streamlined the documentation requirements required for a given acquisition.

The optional practice of conducting a preliminary design review has now been formalized, thus focusing more attention on the requirement definition area of the development effort. This should lessen the problems associated with late requirements identification and configuration control.

A new Software Development Standard (SDS) has been written using MIL-STD-1679 (Navy) as one of its basic documents. The SDS document is at the heart of the development effort since it defines the contractor's responsibilities. It emphasizes sound software engineering practices, such as top-down design, structured programming, and modularization. Other changes to existing standards are being implemented in areas such as Configuration Control, Equipment and Computer Programs, Specification Practices, and Technical Reviews and Audits for Systems, Equipments and Computer Programs. Two documents have been prepared in the area of Quality Assurance: (1) the Software Quality Assurance Measurement (SQAM) document, specifying required measurements, and (2) The Software Quality Policy, detailing the policies governing quality assurance and which will likely replace the current Software Quality Assurance Program Requirements, MIL-STD-52779.

VII. STARS

A. OVERVIEW OF STARS

The scope of the STARS (Software Technology for Adaptable , Reliable Systems) program is perhaps the broadest and farsighted software initiative ever undertaken. It addresses almost every socioeconomic, technological, political, and psychological aspect associated with the problems of software development and maintenance for major military systems. STARS is deliberately structured [Ref. 34: p. 14] to facilitate and encourage the rapid transition of new technology into practice. STARS is intended to be an impetus for a cooperative environment among the governmental, commercial, and academic sectors of U. S. society in which technology transfers will freely occur, and through which highly automated and efficient software support environments will be developed.

The DoD has a continuing interest in the development of computer technology. It is in the best interest of the DoD and the country to maintain a front-runner position in computer technology. To this end, the DoD has established the VHSIC and Ada programs. The VHSIC program (very high speed integrated circuit) aims "to gain and maintain a qualitative lead over potential adversaries by providing affordable complex military functions in extremely small, ultrareliable packages suitable for operation in severe military environments." [Ref. 34 : p. 16] The Ada program entails the development of a high-order language for mission critical computer systems. While both programs have made strides in maintaining American superiority in computer technology, a software initiative is being launched to

complement them. STARS aims to develop the systems and software techniques through which this superiority can be maintained.

DoD has found that software changes are easier and less costly than changes to physical components of military systems. While this can be a major military advantage, the needed technology to make these software changes is not always available. The software requirements are ahead of the systems needed to institute them. Other problems involved in the software dilemma besides inadequate technology include inappropriate acquisition and management practices and a serious shortage of skilled people. Controlling and managing software projects is a major concern of DoD. Costs for software are becoming the major cost factor on many systems projects. These costs must be predicted and controlled. The supply of trained professionals is inadequate. Currently the gap "between demand and supply has been estimated in terms of 50,000 to 100,000 software professionals, and if nothing is done, this gap could become 860,000 to 1,000,000 software professionals by 1990." [Ref. 35 : pp. 52 - 53]

STARS looks at addressing the technology, management, acquisition and personnel problems in two ways which will parallel each other. The long range approach is to "leapfrog current technology and completely change the view of the software process", as quoted from reference 8STAR3. This approach is deemed necessary since current methodologies do not appear to be able to satisfy fully the future requirements. Opportunities on the horizon which are to be evaluated include: expert systems, very high level languages, functional programming and program generation systems. While successful fulfillment of these opportunities will enhance the software environment, they will take time to develop. The second approach is to "bridge the gap" until

the more futuristic opportunities can be developed. The second approach entails an evolutionary strategy of building upon the existing systems, improving them, adding techniques, refining models, and training people along traditional lines of software development. As stated by Boehm and Standish, this approach is necessary to "combat the software supply-demand gap". By learning how to manage skillfully the large number of variables involved in software projects and integrating the key concepts existing in the software environment, managers can utilize their resources needed for effective software development. Completeness and integration are the key concepts of this second approach. [Ref. 36 : pp. 30 - 37]

B. OBJECTIVES

The primary goal of the STARS program is to "improve productivity while achieving greater system reliability and adaptability." [Ref. 35 : p. 56] DoD software in many instances is of vital importance in providing life-essential functions, such as computerized flight controls. Due to this stringent requirement, reliability is of utmost importance. The software must be easily adapted to changes in mission requirements. A third key element is that of affordability. As stated earlier, cost is an important factor and becoming more so as more systems are software dependent. These three items, reliability, adaptability and affordability form the backbone behind the goal of STARS. As stated by the initiating task force of STARS, "We need to improve the state of practice throughout the DoD community so that we can provide development and in-service support that is faster, less expensive, and more predictable and results in software that is more powerful, reliable, and adaptable." [Ref. 35] Based on this goal of an improved

software development environment, three basic objectives are established for STARS: 1) expand the level and base of expertise in both the government and private sector; 2) improve management methods, application-independent-technical, and application-specific tools; and 3) increase the use of tools by adding incentives, improvements to useability and added automation and integration. For each of the objectives, a task area has been established with specific plans of pursuing the objectives. This paper will discuss the task areas of effectiveness measurements, project management and acquisitions.

"The STARS program will be carried out within the context of a variety of on-going and planned activities. It will establish a basis for close coordination, consistency, and commonality while pursuing the additional work that assures the broad scope and clear focus of the overall DoD software program." [Ref. 37 : pp. 21-29]

The program will be instituted in a 7-8 year period. Beginning in FY84 with the preparation stage, the following three consecutive two year periods include the consolidation, enhancement, and transition stages. The consolidation stage focuses on putting current technology into practice. This includes fully utilizing the management tools, automated software tools and implementing the latest procurement strategies. The second stage focuses on enhancing the environment established in the first stage. This is an evolutionary process of refinement and improvement. The final stage will institute a fully funded STARS program. Also in this transitional stage any R&D developments which have reached fruition can be transitioned into utilization in the software environment.

C. ORGANIZATION

The program is vertically managed under the Under Secretary R&D. A joint Service team under the Under Secretary will provide the initial planning and coordinating of the program. Contractors will assist as required and selected as appropriate by various DoD agencies. To aid in the government/contractor/academia interface, a free exchange software engineering institute will be established to encourage technology transfer and thus promote a commonality of goals and interest. The technology transfer will be further enhanced by various DoD agencies' R&D centers concentrating on their particular area of interest rather than attempting to cover the full spectrum of software engineering. Also each DoD agency will be assigned responsibility of supporting various technology areas. Funding for the program is proposed to rise from the \$60M level in FY84 to the \$100M level in FY86 (constant FY84 dollars).

D. EFFECTIVE MEASUREMENTS

Measurement of key elements in a system allow one to understand the system process and therefore control the process [Ref. 38 : pp. 47 - 53]. Maintaining control and predicting outcomes in software development projects is a major advance in software technology. Practical benefits of being able to achieve effective measurements include: 1) provides a description of the software environment; 2) allows possible prediction of project parameters such as cost, delivery time, constraints, and quality; 3) permitting the expression of requirements and goals quantitatively; 4) ability to track progress and provide feedback; and 5) providing a means of analyzing costs and benefits. While these benefits are great, obtaining the ability to have reliable measurements is a task unto itself.

Two areas needing effective measurement are software performance and user performance. Software performance becomes more important as software plays a larger role in the overall system. Software systems must be able to interface and effectively synchronize to function properly. Performance of users has an impact on the cost and time required to produce systems. Studies have shown that developing reliable models to predict such performance is near to impossible.

STARS intends to institute a uniform method of approaching the measurement task. In keeping with the overall goal of STARS, an environment conducive of model and metric development will be evolved. In general terms, the development and refinement of existing models will continue. More data will be gathered and the iterative process of hypothesis testing will continue. There will be a widespread emphasis on using measurement tools and models. Manual as well as automated tools will be made standard as much as possible. With an increasing data base, baselines will be defined and maintained. These baselines will include size, effort, reliability, and the use of methods and tools. All in all the benefits of the measurements will be to allow the assessing of methods and tools in order to get the most product from the least amount of resource expenditure.

E. PROJECT MANAGEMENT

"The primary objectives of the project management task area are: 1) enhance the buyer manager's capability in early project planning; 2) provide a better means of communicating and coordinating between and within buyer and producer organizations; 3) furnishing tools to aid managers in identifying and correcting problems before they affect schedule of functional capability; and 4) increase the availability of software engineers educated in the principles of project management." [Ref. 39 : p. 57]

Most software system development projects involve the DoD and a contractor with the DoD component being the buyer and the contractor being the producer. Early project planning performed by DoD project managers is often lacking. Many projects reach the award stage before proper planning has taken place in the areas of mission analysis, requirements definition, scheduling and cost identification. This causes problems of unspecified work statements and misguidance of contractors in the early contract period. The bottom line is that poor planning costs money. STARS intends to overcome this through general guidelines in the pre-award contract phase.

The second objective deals with communication between the contractor and the government and the overall contracting process. Communications are intended to be improved through better documentation and the building of closer working relations between the contractor and the government. The contracting process will be addressed through the establishment of a software acquisition panel. The panel, made up of various service representatives including STARS and input from industry, will recommend appropriate acquisition policies, contract incentive mechanisms, and make recommendation and promote changes to the software systems acquisition process.

The third objective is to equip the manager with a standard "tool kit" consisting of management tools which will allow identification of problems before they can impact greatly on the project. This tool kit should also be available to the contractor so that communication will be along the same lines. Examples of tools are: data base managers, word processing, telecommunications, graphics, spreadsheets, schedule generator, cost estimation and general reporting systems. The aim of the tools is to automate the tracking of the project.

The final objective is that of educating the project managers into the proper management perspective. This calls for the development of standard job descriptions followed by training in the areas of project management. This objective is important since most individuals involved in project management of software systems were or are software professional and not management professionals.

F. IMPROVING PERSONNEL RESOURCES

Overall, the demand for software is increasing at 12% per year, while the supply of software-producing personnel is increasing at an annual rate of only 4 percent. [Ref. 40: p. 31] If this trend continues, the shortage of software-producing personnel will increase tenfold to an estimated shortage just under one million software professionals by the year 1990. Each of the services have already reported shortages of qualified software personnel and predict that these shortages will become critical by the late 1980's. [Ref. 35: p. 53] Another area of concern is maintaining the skill levels of present software personnel abreast of the skill level demanded by rapidly changing technology.

The task objective to improve personnel resources is based on two fundamental premises: (1) increasing the level of expertise, and (2) expanding the base of expertise available to DoD. The strategy and major subtasks for achieving this objective is presented in detail in the article by Orglesby and Urban [Ref. 41 : pp. 65 - 70] and will now be highlighted.

1. Key Population Assessment

This major subtask is designed to assess the human resource issues of the availability, the utilization and the

future requirements of software-related skills. Only through these assessments can skill requirements for software-related skills be determined. Quantitative measurements based on educational units and/or task period performance would then be used to for qualification and classification of employment and career development of software professionals.

2. Career Structures and Incentives

Once the key population assessment is completed and skill requirements known, career structures (career ladders) can be developed and put into place for each of the occupational subspecialties within the software-development field.

3. Exchange Programs

This subtask is structured to increase the number of software personnel exchanges for prescribed periods among government, industry, and academia. Regulations are already in place permitting personnel exchanges between the services and between DoD and state organizations. These established exchange programs are to be better publicized and supported. Exchange programs will be initiated with industry, DoD and academia. These programs offer an excellent medium for technology transfer, training, and a better understanding of the problems associated with a counterpart community, be it inside or outside of DoD.

4. Other Educational Subtasks

Other educational subtasks contemplated under STARS to improve human resources include: (1) academic programs that will encourage the development or enlargement of software engineering programs in colleges and universities, (2) training programs utilizing governmental or nongovernmental programs to advance the educational technology in software

engineering with efforts oriented toward Ada technology, and
(3) learning aids that focus on automated instructional systems and knowledge-based tutorial systems.

G. IMPROVING PROCESSING TECHNOLOGY

The second approach taken by STARS to help develop a software support environment is through the improvement of processing technologies. Processing technology includes the "techniques, methods, practices, and tools supporting software over its complete life cycle". [Ref. 37 : p. 22]

One way in which this objective can be met is through improved application-independent technical tools. These are tools that support projects of all types, regardless of application. Examples of application-independent tools include operating systems, linkers, loaders, compilers, and programming languages. An example of the latter is Ada, which is the cornerstone of current efforts directed toward the development of the Ada Programming Support Environment (APSE). The long-term objective of APSE is to provide a common high-order language through which programming support environment tools can be interfaced. However, for the short-term it is necessary for APSE to accommodate the multilingual inheritance of DoD's diverse, programming-support tool inventory. [Ref. 42 : p. 15]

A second way in which this objective can be achieved is through improved application-dependent technical methods and tools. [Ref. 34] Examples of this category of tools include Very High Level Languages (VHLL), libraries, test drivers, and simulators.

Mid-- to long-term objectives of these application-specific task areas involve the use of emerging technology, such as VHLLs, Knowledge-based systems, and program generators. The short-- to near-term objectives (next seven years)

of this task are centered around the software "reusability" problem in which software for each new system has been developed in total, from-the-ground-up, as though it is the first and last system of its kind. Future efforts will be directed toward the development of Ada-based reusable software. Reusable software is hardly a new idea, but past attempts to create sets of reusable software have failed for lack of quality control. To overcome similar problems, DoD's software must be developed with the following characteristics: [Ref. 37 : p. 79]

- precise statements and validations of module functions and interfaces.
- generalized performance functions to increase scope of application.
- use of high programming standards and widely-accepted programming methodologies.
- robust behavior. Not only must software be reusable, reusable software must also be accessible by software developers. Techniques for cataloging and warehousing reusable software must also be implemented. Current data base management techniques for the query, management, and retrieval are considered appropriate for this application. [Ref. 43 : p. 18]

H. INCREASING USE OF PROCESSING TECHNOLOGIES

Improved processing technology for software development can only make a difference if people use this improved technology. Another objective of the STARS program is to increase the appropriate use of these technologies. Two of the subtask area insupporting this objective are (1) improve business practices, and (2) improve tool usability.

1. Improve Business Practices

This subtask is aimed at changing current DoD regulations in order to facilitate the acquisition of software. Another goal of this subtask is to utilize financial incentive schemes to encourage capital investment by industry

directed at the coordinated pursuit of new technology development.

2. Improve Tool Usability

This subtask focuses on improving the interaction between computer-based systems and the users or developers of software. In her article, [Ref. 19] Elizabeth Kruesi lists three basic objectives in this area:

- to expand the technology base by supporting the continued development of knowledge, methods, and tools for incorporating human factor concerns into system development,
- to expand the experience base through the application of this technology to actual development projects, and
- to ensure effective human factor engineering of automated programming support environments by focusing on the special needs of software professionals.

Although Kruesi suggests many techniques and methods for improving tool usability (such as defining user interface goals, early user testing, predictive tools for interface design, and following proven interface-design guidelines), she sees imperical testing as the one human engineering method offering the most promise. Although noticeably lacking in the past development of tools and environments for software personnel, imperical testing is viewed by Kruesi as an especially "rich source of ideas for user interfaces, particularly in the design of advanced software environments such as Smalltalk and Interlisp...."

Although one of the benefits that will be realized from the improvement of tool usability is increased productivity, the primary benefit may very well be in the avoidance of human error in the design, development, and maintenance of life-dependent and mission-critical DoD systems.

I. CONCLUSION

This chapter has presented many of the managerial and technologically-oriented objectives that DoD has incorporated under the STARS program. This software incentive is enormously broad in its scope, including all major sectors of society in both present and future efforts to keep this country at the forefront of software technology. Although still in its infancy, STARS has defined many existing software problems and has established both evolutionary and revolutionary strategies to minimize these problems in the future. The conceptual foundation of STARS is sound and promises to improve future software development should the program receive the financial support that it deserves.

STARS is an aggressive approach to a well defined set of problems. The key to the success of STARS, as is true of any government initiative, is the widespread acceptance of the concepts surrounding it. The key element driving STARS is that of standardization as supported through commonality of methodology, uniform metrics and baselines. The software institute calls for a sharing of information and the ever-increasing technology transfer.

VIII. CONCLUSIONS AND RECOMMENDATIONS

The gains made in software engineering over the past two decades have been significant, yet software projects failures continue with alarming regularity as both the size and complexity of computer-based systems continue to grow.

There is no shortage of proposals to confront the problems that plague the development and acquisition of software. Yet, the very nature of software continues to defy its quantitative analysis resulting in obscured visibility and ineffective controls in the development and maintenance process.

Although great strides have taken place in the formulation of software metrics as management information tools and as a medium to provide feedback to software engineers, attempts to devise metrics to quantify software quality have remained elusive. Software quality assurance programs, such as described in MIL-STD-52779(A), provide a planned and systematic approach for building quality into software. Successful implementation of these programs have given credence to the saying the "quality is free," in the long run primarily through cost savings inherent in truly maintainable software.

Software maintenance is the neglected phase in the software life-cycle. Maintenance accounts for well-over half of all resources expended on software throughout its life. The trend in the amount of efforts needed to maintain software is increasing at a dramatic rates, consuming resources that were once reserved for developmental efforts. Yet, project management does not often give sufficient consideration to building maintainability into software as an indispensable criteria in the design process. Various technical and

managerial approaches can be implemented within the maintenance activity with minimum upheaval, but the most influential factors leading to the maintainability of software occur during the phases prior to the maintenance phase.

Chapter VI focuses on various high-level efforts directed at the standardization of computer technology and software within DoD. If the appropriate selection of tools, methods, and methodologies advocated by current software literature and directives were to be put into practice, better software would be realized in DoD. In order to assure the success of this undertaking, it has been suggested by numerous authors that the project manager should be provided with sufficient technical background. This approach is the likeliest to assure failure. Today, the framework in which DoD software is acquired and developed is both disjointed and perplexing. The myriad of instructions and guidelines offer platform of confusion not resolution. Significant progress has been made by groups such as the JCL in attempting to standardize, through joint-service instructions, many of the aspects affecting the acquisition and maintenance of software. Much remains to be done.

The immaturity of the software engineering discipline has been too often been pointed to as the primary culprit of software failure. Software engineering must never mature; it must continue to evolve at a pace set by advances in our technological society. Software engineering is but one of the factors contributing to the delivery of quality software. Standardization is the key in reversing the trend of the delivery of overbudget, overschedule, inferior software.

The management and development of software today is like trying to understand a United Nations assembly without interpreters. Today there are far more programming languages than there are different languages at the United Nations, with revisions bastardizing the integrity of its

parent programming language as dialects bastardize their mother language. Yet programming languages is just one aspect of the total standardization effort that must take place in DoD. The best of today's management systems can be consolidated into a single, joint-service system understood by management personnel in both DoD and industry. The adoption of any one set of tools, methods, and methodologies for the development and acquisition of software is far better than attempting to live by all of the sets available.

Benefits derived through standardization should be exploited to their fullest. The broadest and most farsighted of these efforts is the STARS program, which addresses most aspects that define the total software life-cycle environment.

APPENDIX A

GLOSSARY OF SOFTWARE QUALITY ATTRIBUTES

Definitions provided in this appendix are derived from [Ref. 18 : Appendix B] and [Ref. 16 : pp. 3-4 - 3-24].

ACCESSIBILITY: Code possesses the attribute accessibility to the degree that it facilitates selective use of its parts.

ACCOUNTABILITY: Code possesses the attribute accountability to the degree that its usage can be measured.

ACCURACY: Code possesses the attribute accuracy to the degree that its outputs are sufficiently precise to satisfy their intended use.

AUGMENTABILITY: Code possesses the attribute augmentability to the degree that it can easily accommodate expansion in component computational functions of data storage requirements.

COMMUNICATIVENESS: Code possesses the attribute communicativeness to the degree that it facilitates the specification of inputs and provides outputs whose form and content are easy to assimilate.

COMPLETENESS: Code possesses the attribute completeness to the degree that its parts are present and each part is fully developed.

This implies that external references are available and required functions are coded and present as designed.

CONCISENESS: Code possesses the attribute conciseness to the degree that excessive information is not present.

CONSISTENCY: Code possesses the attribute consistency to the degree that it contains uniform notation, terminology, and symbology within itself, and external consistency to the degree that the content is traceable to the requirement.

DEVICE EFFICIENCY: Code possesses this attribute to the degree that the operation, function, or instructions provided by the code are performed without waste of resources with respect to that device.

DEVICE-INDEPENDENCE: Code possesses this attribute to the degree that it can be executed on computer hardware configurations other than the current one.

EFFICIENCY: Code possesses this attribute to the degree that it fulfills its purpose without waste of resources.

HUMAN ENGINEERING: Code possesses this attribute to the degree that it fulfills its purpose without wasting the user's time and energy, or degrading their morale.

LEGIBILITY: Code possesses this attribute to the degree that it is easily discerned by reading the code.

MAINTAINABILITY: Code possesses this attribute to the degree that it facilitates updating to satisfy new requirements or to correct deficiencies.

MODIFIABILITY: Code possesses this attribute to the degree that it facilitates the incorporation of changes, once the nature of the desired change has been determined.

PORTABILITY: Code possesses this attribute to the degree that it can be operated easily and well on computer configurations other than its current one.

RELIABILITY: Code possesses this attribute to the degree that it can be expected to perform its intended functions satisfactorily.

ROBUSTNESS: Code possesses this attribute to the degree that it can continue to perform despite some violation of the assumptions in its specifications.

SELF-CONTAINEDNESS: Code possesses this attribute to the degree that it performs all its explicit and implicit functions within itself.

SELF-DESCRIPTIVENESS: Code possesses this attribute to the degree that it contains enough information for a reader to determine and verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status.

STRUCTUREDNESS: Code possesses this attribute to the degree that it contains a definite pattern of organization of its interdependent parts.

TESTABILITY: Code possesses this attribute to the degree that it facilitates the establishment of verification criteria and supports evaluation of its performance.

UNDERSTANDABILITY: Code possesses this attribute to the degree that its purpose is clear to its inspector.

USABILITY: Code possesses this attribute to the degree that it is reliable, efficient, and human-engineered.

APPENDIX B

HALSTEAD AND MCCABE'S SOFTWARE METRICS

As generally addressed in Chapter III, this appendix will present quantifiable measurements of various software characteristics using both Halstead's Software Science Theory and McCabe's Complexity Measure.

A. HALSTEAD'S SOFTWARE SCIENCE

Halstead begins with four basic metrics:

1. n_1 --the number of unique or distinct operators in the program.
2. n_2 --the number of unique or distinct operands in the program.
3. N_1 --the total usage of all the operators in the program.
4. N_2 --the total usage of all the operands in the program.

Tables 7 and 8 show the resulting counts of the operators and operands used the algorithm used in table 6 . [Ref. 10 : pp. 3 - 18]

The vocabulary (n) of a module is described as the sum of its unique operators and operands:

$$n = n_1 + n_2$$

TABLE 6
A Sorting Subroutine

```

SUBROUTINE SORT (S,N)
DIMENSION X(N)
IF (N.LT. 2) RETURN
DO 20 I = 2,N
    DO 10 J = 1,I
        IF (X(I).GE. X(J)) GO TO 10
        SAVE = X(I)
        X(I) = X(J)
        X(J) = SAVE
    10 CONTINUE
    20 CONTINUE
RETURN
END

```

TABLE 7
Operator Count

Operators	Count
1. End of statement	7
2. Array subscript	6
3. =	5
4. IF ()	2
5. DO	2
6. End of program	2
7. .LT.	1
8. .GE.	1
9. GO TO	1
10. GO TO	1
n1 = 10	N1 = 28

Similarly, the length (N) of a module is defined as the sum of all operators and operands used in the module:

$$N = N1 + N2$$

TABLE 8
Operand Count

Operands		Count
1.	X	6
2.	I	5
3.	J	4
4.	N	3
5.	2	2
6.	SAVE	2
7.	1	1
-----		-----
n2 = 7		N2 = 22

Halstead also introduced a formula for the estimated length (NH):

$$NH = n1 \log_2 n1 + n2 \log_2 n2$$

The estimated length equation (NH) has proven an acceptable estimator for the length, N, of a module. The usefulness of NH seems to be somewhat sensitive to actual program length. Test have shown that this formula works best for programs if N is in the range between 2000 and 4000. In this light, errors can minimized by breaking down modules to where they are within these parameters [Ref. 11]. Halstead attributes this finding to the presence of "impurities" in the program requiring optimization.

When compared against the actual length of the above listed algorithm one finds that N = 50 and NH = 52.9, a difference of less than 5.8% in this particular case.

Additional metrics were defined by Halstead using the terms already presented. Of interest is another measure of program size called volume, (V), which is measured in bits:

$$V = N \times \log_2 n$$

Volume may also be interpreted as the number of mental comparisons needed to write a program of length N, assuming a binary search method is used to select a member of the vocabulary size n. The most succinct form in which an algorithm can be expressed requires prior existence of a language in which the required operation has already been defined or implemented. In such a case, the implementation of that algorithm would require no more than naming of operands for its arguments and its resultants (eg. SORT(X)). These algorithms are considered minimal in size and are said to have the potential volume V^* :

$$V^* = (2 + n_2) \times \log_2 (2 + n_2^*)$$

(where n_2^* is the different input and output parameters)

Any program with volume V is considered to be implemented at the program level, L, defined as:

$$L = V^*/V$$

Notice that for the most succinct version of any algorithm, the resultant is 1. As the unique operators (n_1) increase and the reuse of operands (N_2) increases the resultant approaches 0. The term "difficulty" is derived from the logic that as the volume of a program increases, the program level (L) decreases and the difficulty increases. Thus, difficulty (D) is the inverse of the program level

$$D = 1/L$$

Since the volume (V) is the number of mental comparisons, and the difficulty (D) is the measure of the average elementary mental discrimination required for each mental comparison, then by combining the formulas for L and D, the total number of elementary mental discriminations, effort, (E), required to generate a program can be derived from

$$E = V/L$$

The advantage of Holstead's measurement of effort (E) is a significant break from traditional use of LOC's. The use of LCC's required the collection of data and regression analysis. Only the number and use of operators and operands are needed to derive measurements for effort, thus overcoming the forementioned difficulties of using LOC's methodology. Another advantage of using E is that it is a strong indicator of the comprehensibility of a program and its propensity for errors [Ref. 10 : p. 16] and [Ref. 11 : p. 34].

Exploring the formula for L further, it can be noted that as the potential volume (V*) increases, the program level (L) decreases proportionately. Consequently, the product L times V* remains constant for any one language. This product, the language level, which is denoted as LAMBDA, is derived by the following formula

$$\text{LAMBDA} = L \times V^*$$

The last of Holstead's formulas to be discussed is used to measure the programming time (TH) for a program in seconds. Halstead adopted the concept introduced by John Stroud, a psychologist, who defined a "moment" as the time required by a human brain to perform the most elementary discrimination. These "moments" according to Stroud, occur at a rate of 5 to slightly less than 20 per second. Halstead then determined that the programming time of a program could mathematically be defined as

$$\text{TH} = E/S$$

(where S = the "Stroud" number = 18)

Halstead reason for selecting "18" for the value of the Stroud number remains a mystery, but it fits his formula nicely and is likely to remain unchallenged until the disciplines of cognitive psychology and software science merge.

B. MCCABES'S COMPLEXITY MEASURE

A program graph is used to represent control flow, as

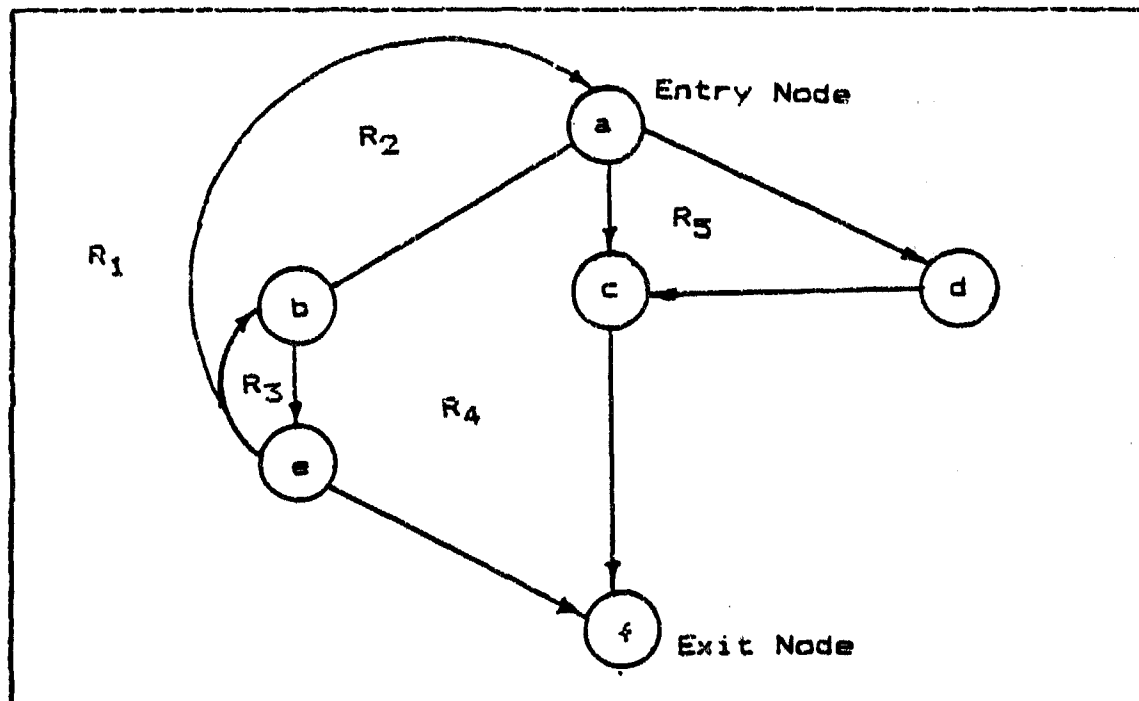


Figure B.1 Control Flow Graph Complexity

illustrated in Figure B.1 [Ref. 12 : PP. 308 -320] The circles represent processing tasks, which can be one or more source code statements. Arrows depict control flow (branching) between processes. Thus, in Figure B.1 , process "a" may be followed by process "b," "c," or "d," depending on which condition was satisfied in process "a." The control flows depicted by arrows going from process "e" to processes "b" and "c" represent "backward" branching. "Regions" may be described as the enclosed areas on the plain of the graph represented by R1 through R5 in Figure MCCABEFIG. These regions represent the bounded areas within the program graph, as well as the unbounded area outside of the graph.

McCabe uses a software complexity measure that is based on what he terms the "cyclomatic complexity" of a program graph for a module. One approach that can be applied in determining the cyclomatic complexity, $V(G)$, is by calculating the numbers of regions in a planar graph. In Figure B.1, for instance, $V(G)$ is equal to 5. Another method is through the formula

$$V(G) = e - n + 2p$$

in which "e" is equal to the number of "edges," i.e. number of arrows, "n" is equal to the number of "vertices," or processes, and "p" is equal to the number of connected components. In Figure B.1, the values of these elements are:

$$n = 6 \text{ (a, b, c, d, e, f)}$$

$$e = 9 \text{ (a to b, a to c, a to d, b to e, e to b, e to a, d to c, e to f, c to f)}$$

$$p = 1$$

By inserting these values into the above formula, the resulting cyclomatic complexity metric, $V(G)$ is again equal five. McCabe also contends that the $V(G)$ measure can provide a quantitative indication of the maximum size, testing difficulty, and reliability of a module. Through empirical investigation, he has found that a cyclomatic complexity measurement of 10 to be a practical upper limit for module size. Exceeding this upper limit makes it increasingly difficult to adequately test a module.

APPENDIX C

STRUCTURED METHODOLOGIES

As discussed in Chapter V, the design of maintainable software is based on the application of a set of engineering principles and practices that include:

- Structured analysis
- Structured design
- Structured programming
- Program design language

This appendix presents the detailed characteristics of these elements as provided in [Ref. 26 : Appendix A].

A. STRUCTURED ANALYSIS

When applied to software development, the characteristics of structured analysis and the logical model are as follows:

- A system is described by the systematic decomposition of broad system functions into subfunctions of progressively finer detail.
- Each function and subfunction is defined by describing
 - Its inputs and outputs
 - Processing activities and requirements
 - Nontransient data stored by the function
- Functions and subfunctions are analyzed to access
 - The support of functions by hardware and software
 - Algorithm and computational requirements (Function, precision, range, timing, etc.)
 - The need for performance and tradeoff studies
- Stored and interface data are analyzed to access
 - Access requirements
 - Structured, format, and storage requirements.

- Functions and data are analyzed to evaluate the requirements for execution and support software.

The net result of the this structured analysis process should be a logical model that defines the complete system which reflects all facets of the system specification and software requirement document. The model should be a form of communication easily understood by both technical and nontechnical personnel, alike. Through the use of this model, the system analyst should be able to develop system requirements without undue consideration of physical implementation constraints. On the other hand, nontechnical users should readily be able understand how the required functions fit together within the context of the whole system.

B. STRUCTURED DESIGN

Structured design is the process of subdividing the software design into hierarchical in a manner that tends to maximize module independence. Benefits provided to the development and maintenance include increased understandability of the system and a minimization of the expenses associated alteration of the software.

The structured design approach has the following characteristics:

- Hierarchical functional tree charts are developed showing a series of boxes representing descending levels of functions and subfunctions. These charts depict the functionality in the same sense that a project work breakdown structure (WBS) depicts hierarchies of work to be performed by a contractor.
- Modularity of components is emphasized. A key characteristic of modularity is a maximum independence of one component from others. This independence allows for a concentration on definition of inputs, outputs and processing of each component. It also facilitates design clarity, which facilitates future modifications.
- At each level of component design, strong emphasis is placed on defining inputs, outputs, and processing of each component. This emphasis represents a key characteristic of both structured design and analysis.

C. STRUCTURED PROGRAMMING

Structured programming requires that a programmer use only a limited set of three basic program structures. These are depicted in Figure 4.3, and are as follows:

- Sequence of two or more operations
- Conditional branch to one of two operations and return (IF a THEN b ELSE c)
- Repetition of an operation (DO WHILE)

Any program, regardless of its complexity, can be implemented using these basic program structures. The use of only these structures limit the procedural design of a program to a small amount of predictable operations. It should be noted, however, that use of only these three structure may lead to inefficiencies in situations such as when an escape from a set of nested conditions or loops is needed. In situations such as these, the designer is left with the options of re-design to avoid these conditions or allow for deviation from these basic structures in a controlled manner.

Two extensions to the basic structures, also illustrated in Figure 4.2, are the DO UNTIL and CASE structures. These are special cases of the other structure which improve both readability and source code without degrading programming structure.

D. PROGRAM DESIGN LANGUAGE

A program design language is basically a test processor that is used to document a structured design. It has the following two characteristics:

- It produces an English-like representation of components of code that are easy to read and comprehend.
- It is structured in the sense that it uses structured programming constructs to show nested logic.

LIST OF REFERENCES

1. Pressman, H. G., Software Engineering: A Practitioner's Approach, McGraw-Hill, 1982.
2. Glaseman, S., Comparative Studies in Software Acquisition, Lexington Books, 1982.
3. Lehman, M.M., "The Environment of Program Development and Maintenance," Proceedings, 1981 International Computer Symposium, 1981.
4. Schnidler, Max, "Defense Dept. Looks to STARS for Better Software," Electronic Design, 12 May, 1983.
5. General Accounting Office Report FGMSD-80-4, Contracting for Computer Software Development--Serious Problems Require Management Attention to Avoid Wasting Millions, November, 1979.
6. Naval Research Laboratory Report 7909, The MUDD Report: A Case Study Of Navy Software Development Practices, by David M. Weiss, May 27, 1975.
7. Technical Management Committee of the Aerospace Industry Association, "Suggestions for DoD Management of Computer Software," Concepts, vol. 5, no. 5, Autumn, 1982.
8. DeMarco, T., Controlling Software Projects, Yourdon Press, 1982.
9. Halstead, M. H., Elements of Software Science, North-Holland, 1977.
10. Fitzsimmons, A. B. and Love, L. T., "A Review and Evaluation of Software Science," ACM Computer Surveys, vol. 10, March, 1978.
11. Conte, S. D., Dunsmore, H. E., and Shen, V. Y., "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," IEEE Transactions on Software Engineering, vol. SE-9, no. 2, March, 1983.
12. McCabe, T. J., "A Software Complexity Measure," IEEE Transactions on Software Engineering, vol. 2, December, 1976.

13. Electronic Systems Division, AFSC Report ESD-TR-76-271, A Review of Software Cost Estimation Methods, by MITRE Corporation, August, 1976.
14. Davis, R. M., "Reducing Software Management Risks," Defense Systems Management Review, vol. 1, no. 6, Summer, 1978.
15. Knight, B. M., "Software Quality and Productivity," Defense Systems Management Review, vol. 1, no. 7-8, Autumn, 1978.
16. Boehm, B. W., and others, Characteristics of Quality Software, vol. 1, North-Holland, 1978.
17. Pfau, P., "Applied Quality Assurance Methodology," Proceedings of the Software Quality and Assurance Workshop, ACM, November, 1978.
18. Aeronautical Systems Division (U. S. A. F.) Report ASD TR-78-8, Airborne Systems Software Acquisition Engineering Guidebook for Quality Assurance, by TRW Defense and Space Systems Group, November, 1977.
19. Buck, R. D., and Dobbins, J. A., "Software Quality Assurance," Concepts, vol. 5, no. 4, Defense System Management College, Autumn, 1982.
20. Willoughby, W. J., "Software Reliability by Design: A Critical Need," Defense Systems Management Review, vol. 1, no. 6, Summer, 1978.
21. Canning, R., "The Maintenance 'Iceberg'," IDP Analyzer, vol. 10, no. 10, October, 1972.
22. Naval Postgraduate School Report NPS-54-82-002, Software Maintenance: Improvement Through Better Development Standards and Documentation, by N. F. Schneidewind, February, 1982.
23. B. W. Boehms, "The High Cost of Software," Practical Strategies for Developing Large Software Systems, 1975.
24. Aeronautical System Division (U. S. A. F.) Report ASD-TR-78-43, Computer Program Maintenance, December, 1977.
25. Scherville, I., Software Engineering, Addison-Wesley, 1982.
26. Aeronautical System Division (U.S.A.F.) Report ASD-TR-80-5028, Airborne System Acquisition Engineering Guidebook for Supportable Software, October, 1980.

27. Assistant Secretary of Defense Report APL/JHU SR 75-3, DoD Weapon System Software Management Study, by Kossiakoff, A., and others, June, 1975.
28. Frost and Sullivan, Inc., The Military Software Market in the U. S., vol. 1, June, 1979.
29. Becker, L. G., "Military Computers in Transisiton: Standards and Strategy, Concepts, vol. 5, no. 4, Autumn, 1982.
30. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation and Implementation, Dryden Press, 1983.
31. Klucas, C. H. and others, "Joint Service Software Policy and Standards," Concepts, vol. 5, no. 4, Defense Systems Management College, Autumn, 1982.
32. Air Force Logistic Command, Final Report of the Joint Logistics Commanders Software Workshop, vol. 1, October, 1979.
33. Marciniak, J. I., "A Perspective on Military Software Standardization Efforts," 1983 Workshop on Software Engineering Standards, IEEE Computer Society, 1983.
34. Martin, Edith W., "The Context of STARS," Computer, vol. 16, no. 11, IEEE, November, 1983.
35. E.W. Martin, "Strategy for a DoD Software Initiative," Computer, vol. 16, no. 3, IEEE, March, 1983.
36. B.W. Boehm, T.A. Standish, "Software Technology in the 1990's: Using an Evolutionary Paradigm," Computer, vol. 16, no. 11, November, 1983.
37. L.E. Druffel, S.T. Redwine, Jr., W.E. Riddle, "The STARS Program: Overview and Rationale," Computer, vol. 16, no. 11, IEEE, November, 1983.
38. J.R. Dunham, E. Kruesi, "The Measurement Task Area," Computer, vol. 16, no. 11, IEEE, November, 1983.
39. H.C. Lubbes, "The Project Management Task Area," Computer, vol. 16, no. 11, IEEE, November, 1983.
40. Boehm, Barry W., and Standish, Thomas A., "Software Technology in the 1990's: Using an Evolutionary Paradigm," Computer, vol. 16, no. 11, IEEE, November, 1983.
41. Orley, Charles E., and Urban, Joseph E., "The Human Resources Task Area," Computer, vol. 16, no. 11, IEEE, November, 1983.

42. Schnidler, Max, "Defense Dept. Looks to STARS for Better Software," Electronic Design, 12 May, 1983.
43. Kruesi, Elizabeth, "The Human Engineering Task Area," Computer, vol. 16, no. 11, IEEE, November, 1983.

INITIAL DISTRIBUTION LIST

	No.	Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2	
2. Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2	
3. Department Chairman, Code 54 Administrative Sciences Department Naval Postgraduate School Monterey, California 93943	2	
4. Professor Norman R. Lyons, Code 54LB Administrative Sciences Department Naval Postgraduate School Monterey, California 93943	1	
5. Commander Dean C. Guyer, USN, Code 54GU Administrative Sciences Department Naval Postgraduate School Monterey, California 93943	1	
6. Computer Technology Programs, Code 37 Computer Science Department Monterey, CA 93943	1	