CROSSTALK

May 2008 **The Journal of Defense Software Engineering** Vol. 21 No. 5

LEAN principles

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAY 2008		**00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 21, Number 5, May 2008	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

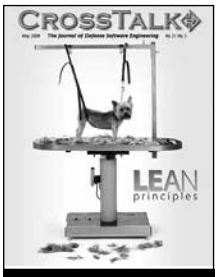
15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Policies, News, and Updates

Lean Principles

Software Engineering Technology



ON THE COVER

Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

Departments

CROSSTALK

CO-SPONSORS:

DoD-CIO	*The Honorable John Grimes*
OSD (AT&L)	*Kristen Baldwin*
NAVAIR	*Jeff Schwalb*
76 SMXG	*Phil Perkins*
309 SMXG	*Karl Rogers*
DHS	*Joe Jarzombek*

STAFF:

MANAGING DIRECTOR	*Brent Baxter*
PUBLISHER	*Elizabeth Starrett*
MANAGING EDITOR	*Ken Davies*
ASSOCIATE EDITOR	*Chelene Fortier-Lozancich*
ARTICLE COORDINATOR	*Nicole Kentta*
PHONE	(801) 775-5555
E-MAIL	crosstalk.staff@hill.af.mil
CROSSTALK ONLINE	www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the Department of Defense Chief Information Office (DoD-CIO); the Office of the Secretary of Defense (OSD) Acquisition, Technology and Logistics (AT&L); U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). DoD-CIO co-sponsor: Assistant Secretary of Defense (Networks and Information Integration). OSD (AT&L) co-sponsor: Software Engineering and System Assurance. USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG); and Ogden-ALC 309 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 20.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

The Chief Cause of Problems? Solutions

The other day I ran across a quote from Eric Sevareid who was a CBS news journalist from 1939 to 1977. He said, "The chief cause of problems is solutions." I do not know the original context of this remark, but it does appear to apply where I work. You would like to think that coming up with solutions gives you a leg up on the problems you face every day, but you wonder sometimes if we are making any headway here? In a very complex organization such as the Air Force Material Command, even with the very best intentions you have hundreds of managers and process players constantly adjusting, patching, and modifying the global business operations – the resultant consequences have a ripple effect on many other related business components throughout this command entity. Is anyone watching the real cost to our warfighters by the unifying impact of all these *solutions*?

The heart of Lean is the determination of value. The thing I like about Lean is the idea of stepping back and looking at the big picture of your business operation and making sure you understand which work activities are really making a contribution to the bottom line. Besides the obvious review of the critical work path – where we can focus on reducing waste or improving efficiency – what is also identified are the *targets of opportunity* for elimination, modification, and/or mitigation with regard to reducing the costs to executing your overall mission. The scope of your review can be either large or small; however, I encourage you to not only think about the core service or product you provide, but to also consider all the other surrounding business activities that affect your team. The aforementioned managers and process players trying to help can become another distraction from the mission. Lean provides the perspective and tools to focus, find, and justify the elimination of the non-added value *solutions*.

Jim York provides a similar discussion in *The Way We See the Problem Is the Problem*. Organizations often need to re-look at their situation from a totally new perspective to decide how to improve. This is not easy, but the ones that do it successfully are leaders in their field. In *Welcoming Software Into the Industrial Fold*, James M. Sutton discusses the benefits realized by implementers of Lean methods. We next get a discussion of Lean in practice with Paul E. McMahon's article *Are the Right People Measuring the Right Things? A Lean Path to Achieving Business Objectives*. Bas Vodde discusses one potential activity towards a Lean environment in *Measuring Continuous Integration Capability*. Dr. Alistair Cockburn completes this month's issue by bringing the specifics of incremental and iterative software development to the forefront. His article, *Using Both Incremental and Iterative Development*, defines these two methods, discusses their differences, and explains how they can be used together for everyone's benefit.

As we have used the Capability Maturity Models over the years, we believe in the value of process improvement efforts and our managers have been great champions. As we implemented our process improvement efforts that are now showing benefits to the warfighters, we have also had our stumbles along the way with false starts that resulted in more costs than benefits. The key to true process improvement is to set meaningful goals and then focus on activities that directly contribute to those goals. One goal as a sponsor of CROSSTALK is to provide readers with tools that will enable them to also provide the best value to the warfighters. I hope you benefit from the tools in this issue.

Ronald Wallman
Oklahoma City Air Logistics Center

Mission Impact of Foreign Influence on DoD Software

Defense Science Board Task Force

The Defense Science Board task force assessed the Department of Defense's (DoD) dependence on software of foreign origin and the risks involved. The task force considered issues with supply chain management; techniques and tools to mitigate adversarial threats; software assurance within current DoD programs; and assurance standards within industry, academia, and government. This executive summary highlights the future U.S. ability to ensure and maintain a trusted supply of software to the DoD and the U.S. government. The full report states that there is no absolute guarantee that software can be sanitized of all vulnerabilities, intended or unintended, and recommends a suite of processes and mitigation strategies to reduce the risk of interrupted systems performance and ensure mission success.

Software has become the central ingredient of the information age, increasing productivity, facilitating the storage and transfer of information, and enabling functionality in almost every realm of human endeavor. However, as it improves the DoD capability, it increases the DoD's dependency. Each year the DoD depends more on software for its administration and for the planning and execution of its missions. This growing dependency is a source of weakness exacerbated by the mounting size, complexity and interconnectedness of its software programs. It is only a matter of time before an adversary exploits this weakness at a critical moment in history.

The software industry has become increasingly and irrevocably global. Much of the code is now written outside the United States, some in countries that may have interests inimical to those of the United States. The combination of the DoD's profound and growing dependence upon software and the expanding opportunity for adversaries to introduce malicious code into this software has led to a growing risk to the nation's defense.

A previous report of the Defense Science Board, "High Performance Microchip Supply," discussed a parallel evolution of the microchip industry and its potential impact on U.S. defense capabilities. The parallel is not exact because the microchip fabrication business requires increasingly large capital formation – a considerable barrier to entry by a lesser nation-state. Software development and production, by contrast, has a low investment threshold. It requires only talented people, who increasingly are found outside the United States.

The task force on microchip supply identified two areas of risk in the offshoring of fabrication facilities – that the United States could be denied access to the supply of chips and that there could be malicious modifications in these chips. Because software is so easily reproduced, the former risk is small. The latter risk of *malware*, however, is serious. It is this risk that is discussed at length in this report.

Software that the DoD acquires has been loosely categorized as:

- Commodity products – referred to as commercial off-the-shelf (COTS) software.
- General software developed by or for the U.S. government – referred to as government off-the-shelf software.
- Custom software – generally created for unique defense applications.

The U.S. government is obviously attracted by the first, COTS. It is produced for and sold in a highly competitive marketplace and its development costs are amortized across a large base of consumers. Its functionality continually expands in response to competitive market demands. It is, in a word, a bargain, but it is also most likely to be produced offshore and so presents the greater threat of malicious modification.

There are two distinct kinds of vulnerabilities in software. The first is the common *bug*, an unintentional defect or weakness in the code that opens the door to opportunistic exploitation. The DoD shares these vulnerabilities with all users. However, certain users are *high value targets*, such as the financial sector and the DoD. These high-value targets attract the *high-end* attackers. Moreover, the DoD also may be presumed to attract the most skilled and best-financed attackers – a nation-state adversary or its proxy. These high-end attackers will not be content to exploit opportunistic vulnerabilities which might be fixed and therefore unavailable at a critical juncture. Furthermore, they may seek to implant vulnerability for later exploitation. It is bad enough that this can be done remotely in the internetworked world, but worse when the malefactors are in the DoD's supply chain and are loyal to and working for an adversary nation-state – especially a nation-state that is producing the software that the U.S. government needs. The problem is serious, indeed. Such exploitable vulnerabilities may lie undetected until it is too late.

Unlike previous critical defense technologies which gave the U.S. an edge in the past, such as stealth, the strategic defense initiative, or nuclear weaponry, the U.S. is protected neither by technological secrets nor a high barrier of economic cost. Moreover, the consequences to U.S. defense capabilities could be even more severe than realized. Because of the high degree of interconnectedness of defense systems, penetration of one application could compromise many others.

In a perfect world there would be some automated means for detecting malicious code. Unfortunately, no such capability exists, and the trend is moving inexorably further from it as software becomes ever more complex and adversaries more skilled. Even if malicious code were discovered in advance, attributing it to a specific actor and/or knowing the intent of the actor may be problematic. Malicious code can resemble ordinary coding mistakes and malicious intent may be plausibly denied. The inability to hold an individual accountable weakens deterrence mechanisms, such as the threat of criminal charges, or even separation of the individual or entity from the supply chain.

Task Force Conclusion

The DoD faces a difficult quandary in its software purchases in applying intelligent risk management, trading off the attractive economics of COTS and of custom code written offshore against the risks of encountering malware that could seriously

jeopardize future defense missions. The current systems designs, assurance methodologies, acquisition procedures, and knowledge of adversarial capabilities and intentions are inadequate to the magnitude of the threat.

Task Force Findings

The Industry Situation

The software industry has become increasingly global as suppliers seek lower cost employees, access to a larger talent base, cultures conducive to highly structured processes, and round-the-clock operation. The issue of foreign influence is only one of degree, because many companies develop code in multiple geographic locations and may embed code from other vendors, code from open source developers, or even code of unknown provenance.

While the United States still has preeminence in computer science, Asia is rapidly gaining. The United States retains a pool of talented computer scientists and engineers, but the natural tendency of the industry is to seek the lowest cost supply of talent. In recent years, that has been primarily in India, while China and Russia are on the rise.

DoD's Dependence on Software

In the DoD, the transformational effects of information technology (IT), joined with a culture of information sharing, called Net-Centricity, constitute a powerful force multiplier. DoD has become increasingly dependent for mission-critical functionality upon highly interconnected, globally sourced, information technology of dramatically varying quality, reliability, and trustworthiness.

Software Vulnerabilities

The majority of software used in the DoD are COTS products. Although the DoD takes advantage of the functionality and inexpensive pricing enabled by the huge market, this code has many weaknesses that are exploitable by even moderately capable hackers who have been the beneficiaries of a culture that has produced an evolution of widely disseminated and powerful tools for system intrusion.

The DoD does not fully know when or where intruders may have already gained access to existing computing and communications systems. The Moonlight Maze activities, which are classified and thus not detailed here, and numerous other data points demonstrate that the U.S. government, and specifically the DoD computing systems, is a constant target of foreign exploitation.

The Threat of the Nation-State Adversary

In dealing with a nation-state adversary, the level of threat rises far above that posed by hackers. It can be assumed that the technological capability to craft actionable malicious code mirrors that of the United States' own best computer scientists. Means and opportunity are present throughout the supply chain and life cycle of software development. While code developed in the United States is not immune from risk, the opportunity for an adversary is greatly enhanced by globalization.

A sophisticated adversary would have three possible aims in the exploitation of existing or planted software vulnerabilities: denial of service, stealing of information, and malicious modification of information. The outcome of any of these would also be accompanied by a loss of confidence in the DoD's essential systems.

"In a perfect world there would be some automated means for detecting malicious code. Unfortunately, no such capability exists, and the trend is moving inexorably further from it as software becomes even more complex and adversaries more skilled."

Awareness of the Software Assurance Threat and Risk

The DoD's defensive posture remains inadequately informed of the sophisticated capabilities of nation-state adversaries to exploit globally sourced, ubiquitously interconnected, COTS hardware and software within DoD critical systems. Similarly, decision makers are inadequately informed regarding the potential consequences of system subversion, and the value of mitigating that risk.

The intelligence community does not adequately collect and disseminate intelligence regarding the intents and capabilities of nation-state adversaries to attack and subvert DoD systems and networks through supply chain exploitations, or through other sophisticated techniques.

The DoD does not consistently or adequately analyze and incorporate into its acquisition decisions what supply chain threat information is available.

Status of Software Assurance in the DoD

Software deployed across the DoD continues to contain numerous vulnerabilities and weak information security design characteristics. The DoD and its industry partners spend considerable resources on patch management while gaining only limited improvement in defensive posture.

The evidence gathered during this study was insufficient to quantify the extent to which awareness and protection against the system assurance problem has permeated DoD systems and networks. The panel did, however, identify considerable variation in the extent to which the systems assurance problem is impacting next-generation DoD systems. That impact ranges from extensive with the introduction of internetworked COTS and open source IT into the Army's Future Combat System program, to only slight in the United States Air Force F-22 program.

The DoD defensive efforts, implemented largely through decentralized execution, are difficult to synchronize to achieve a coordinated enterprise effect. The DoD has not effectively allocated assurance resources to address the systems assurance problem, nor has it designed its systems and networks to mitigate this problem in the face of the capabilities of nation-state adversaries.

The primary process relied upon by the DoD for evaluation of the assurance of commercial products today is the Common Criteria (CC) evaluation process. The task force believes that CC is presently inadequate to sufficiently raise the trustworthiness of software products for the DoD. This is particularly true at Evaluation Assurance Level 4 (EAL4) and below, where penetration testing is not performed. Nonetheless, CC evaluation is an international program, well established, and not easy to change.

Ongoing Efforts in Software Assurance

Software assurance is receiving attention at a number of federal agencies and laboratories, including the DoD, National Security Agency (NSA), National Institute of Standards and Technology, and Department of Homeland Security (DHS). Within the DoD, a Software Assurance Tiger Team has been studying

the problem and has developed a comprehensive strategy for managing risk through system engineering, source selection, design, production, and test. The key element of risk management in this strategy is the prioritization of criticality among system components and subcomponents, with special procedures and attention placed on the system components determined to be most critical to mission success.

Supplier Trustworthiness Considerations
It is not currently DoD policy to require any program – even those deemed critical by dint of a Mission Assurance Category I status – to conduct a counterintelligence review of its major suppliers unless classified information is involved. Supplier trustworthiness enters into existing DoD acquisition processes primarily for protection of classified information and for research technology protection. From a systems assurance perspective, supplier trustworthiness should consider adversarial control and influence of the business or engineering processes of the supplier, as well as the ability of the business and engineering processes to prevent outside penetration.

Finding Malicious Code
The problem of detecting vulnerabilities is deeply complex, and there is no silver bullet on the horizon. Once malicious code has been implanted by a capable adversary, it is unlikely to be detected by subsequent testing. A number of software tools have been developed commercially to test code for vulnerabilities, and these tools have been improving rapidly in recent years. Current tools find about one-third of the bugs prior to deployment that are ever found subsequently, and the rate of false positives is about equal to that of true positives. However, it is the opinion of the task force that unless a major breakthrough occurs, it is unlikely that any tool in the foreseeable future will find more than half of the suspect code. Moreover, it can be assumed that the adversary has the same tools; therefore, it is likely the malicious code would be constructed to pass undetected by these tools.

The task force believes that the academic curriculum in computer science does not stress adequately practices for quality and security, and that many programmers do not have a defensive mindset. While many vendors methodically check and test code, they are looking for unintentional defects, rather than malicious alterations.

Government Access to Source Code
It is tempting to consider having the United States government take the source code of a commercial product and run its own vulnerability assessment tools against it. However, there are a number of legal, ethical, and economic barriers that make this an unattractive proposition, particularly from the point of view of the vendor. License agreements forbid reverse engineering of source code, vendors worry about the loss of intellectual property, and perhaps most importantly, they worry about the cost of supporting the actions and findings of a team of outsiders not familiar with the design and implementation of such hugely complex programs. Some of these worries are lessened when the testing is done by an independent laboratory.

"There is a natural tension between the U.S. government's need to know the security worthiness of what they procure and a vendor's need to avoid disclosing particular vulnerabilities."

Conclusion
All of the considerations just listed seem to point to an intractable problem. The nation's defense is dependent upon software that is growing exponentially in size and complexity, and an increasing percentage of this software is being written offshore within easy reach of potential adversaries. That software presents a tempting target for a nation-state adversary. Malicious code could be introduced inexpensively, would be almost impossible to detect, and could be used later to get access to defense systems in order to deny service, to steal information, or to modify critical data. Even if the malware were to be discovered, attribution and intent would be difficult to prove, so the risk for the attacker would be small.

Against this backdrop of potential disaster, practical experience and belief paint a picture of aggravating and continuous software problems, but not ones that are lethal. However, there are some systems on which, to varying degrees, life depends (e.g., power, health). In this sense, DoD systems are among the most critical because their national security mission is often measured in fatalities, and failures that would be innocuous in another context can be lethal and lead to mission failure.

If the attacker cannot be deterred and its malware cannot be found, what is to be done to provide assurance that DoD software will perform in mission-critical situations? Although there never will be an absolute guarantee, software assurance is really not about absolute guarantees but rather intelligent risk management. The risk of vulnerable software can be managed through a suite of processes and mitigation strategies detailed in the Task Force recommendations; this risk can be weighed against the attractive economics and enhanced capabilities of mass-produced, international software.

Task Force Recommendations

Acquisition of COTS and Foreign Software
DoD should continue to procure from, encourage and leverage the largest possible global competitive marketplace consistent with national security.

The DoD must intelligently manage economics and risk. For many applications the inexpensive functionality and ubiquitous compatibility of COTS software make it the right choice. In acquiring custom software the increased risk inherent in software written offshore may sometimes be worth the considerable cost savings. The task force recommends that critical system components be developed only by cleared U.S. citizens.

Increase U.S. Insight Into Capabilities and Intentions of Adversaries
The intelligence community should be tasked to collect and disseminate intelligence regarding the intents and capabilities of adversaries, particularly nation-state adversaries, to attack and subvert DoD systems and networks through supply chain exploitations, or through other sophisticated techniques.

DoD should increase knowledge and awareness among its cyber-defense and acquisition communities of the capabilities and intent of nation-state adversaries.

Offensive Strategies Can Complement Defensive Strategies
The United States government should link cyber defensive and offensive opera-

tions to its broader national deterrence strategies, communications and operations, treating adversarial cyber operations that damage United States information systems and networks as events warranting a balanced, full-spectrum response.

System Engineering and Architecture for Assurance

The DoD should allocate assurance resources among acquisition programs at the architecture level based upon mission impact of system failure. The task force endorses the strategy and methods to accomplish this as developed by the DoD Software Assurance Tiger Team and validated by the Committee on National Security Systems (CNSS) Global IT Working Group.

The DoD cannot cost effectively achieve a uniformly high degree of assurance for all the functionality it uses across many and varied mission activities. Allocating criticality of function levies a requirement for assurance of that function and also of those functions that defend it. Systems identified as critical must then allocate criticality at the subsystem and assembly level.

To properly allocate scarce assurance resources, the DoD must allocate criticality at the system-of-systems and enterprise architecture level. This analysis should occur early within the lifecycle, and should render a prioritization decision no later than Acquisition Milestone A to allow programs of record to appropriately respond to their criticality.

Improve the Quality of DoD Software

The DoD can effectively raise the *signal-to-noise ratio* against software attacks by raising the overall quality of the software it acquires. If there were fewer unintentional bugs in software, the visibility of deliberate malware would be increased. While general improvements in information assurance will not, per se, prevent a determined attacker from corrupting the software supply chain, there are several compelling benefits in improving the overall assurance/security worthiness of COTS.

A sophisticated adversary would have to work harder to introduce an exploitable vulnerability instead, as is currently the case, of relying upon the plausible deniability of a common programming error to avoid attribution of malicious intent. Furthermore, a sophisticated adversary would have less confidence that its malware would remain undetected, invisible in a world containing far fewer distracting vulnerabilities. That uncertainty could be a deterrent in itself.

Improve Tools and Technology for Assurance

Improve Trusted Computing Group (TCG) Technologies

The TCG initiatives, centered on the Trusted Platform Module (TPM), provide a means for containing intrusions into separated information domains. Each chipset that implements the TPM embeds a unique identifier. Cryptologic verification of this identity is required when access to system assets is requested. TPM may help ensure that only approved and signed code is run, thus reducing the risk of unapproved code being installed.

The NSA and others have identified a number of improvements and complementary practices that would strengthen TCG-compliant systems, including privacy-preserving attestation, virtualization, and architectures that provide richer software assurance measurement and monitoring capabilities.

Improve Effectiveness of Common Criteria

Currently, the official DoD-wide evaluation/validation scheme is the National Information Assurance Partnership based upon the CC. The reality today is that it would be far easier and more effective to improve CC than to invent a new scheme specific to the DoD or to DHS.

A number of ways to strengthen CC are discussed in the Recommendations section of this report. Among these suggestions are crediting vendors for the effective use of better development processes, including the use of automated vulnerability reduction tools and automated tools for vulnerability analysis during EAL4 and below. Validation schemes should also reduce artificial artifact creation and rely upon artifacts that are generated by the development process.

Improve Usefulness of Assurance Metrics

There is a natural tension between the United States government's need to know the security worthiness of what they procure and a vendor's need to avoid disclosing particular vulnerabilities. One way to satisfy both needs would be to develop a weighted index of the security worthiness of software. A weighted score could be generated via testing based on some combination of the utility of the tools themselves, the amount of code coverage of the tools, and the test results against a particular product. The entire development process should also be evaluated.

More Knowledgeable Acquisition of DoD Software

The DoD should implement a scalable supplier assurance process to assure that critical suppliers are trustworthy. No product evaluation regime in effect today provides insight into a vendor's real development processes and their effectiveness at producing secure and trustworthy software – so the software assurance challenge for the DoD is to define an evaluation regime that is capable of reviewing vendors actual development processes and rendering a judgment about their ability to produce assured software.

The DoD acquisition process should require that products possess assurance matching the criticality of the function delivered. Furthermore, the DoD should require that all components should be supplied by suppliers of commensurate trustworthiness, and in particular, that all custom code written for systems deemed critical be developed by cleared U.S. personnel.

The collective buying power of the United States government is such that it can force change on its suppliers to a degree no other market sector can reasonably do. The DoD, working in collaboration with the Office of Management and Budget, DHS, and other federal agencies, can help to change the market dynamic through both positive and negative incentives so that they get better quality software, and to make better risk-based and *total cost*-based acquisitions.

Research and Development in Software Assurance

The DoD should establish and fund a comprehensive science and technology strategy as well as programs to advance the state-of-the-art in vulnerability detection and mitigation within software and hardware. The goals of the classified and unclassified research and development investments in assurance should be to develop the technology to effectively take accidental vulnerabilities out of systems development and to improve TCG technologies in order to bound most risks of intentionally planted software. This program should monitor what markets are delivering, identify gaps between what the market is delivering and what the DoD needs, and fill the gap.◆

For more information on the Defense Science Board Task Force findings, go to <www.acq.osd.mil/dsb> and search under "Reports."

The Way We See the Problem Is the Problem

Jim York
FoxHedge Ltd

Having helped organizations implement Lean processes over the past eight years, I have had the opportunity to see a similar theme played out many times. Organizations unwilling to alter their way of seeing their problem flounder in their transformation to a Lean enterprise and ultimately regress to their former processes. Resistance to changing their way of seeing the problem is the biggest impediment for organizations trying to achieve success with Lean.

Perhaps the biggest challenge is how organizations think about value. From a Lean perspective, organizations create value when they deliver to customers what they want, when they want it, and at a price they're willing to pay [1].

For almost 100 years, the American industry's perception of value has been colored by the teachings of scientific management made popular by Frederick Winslow Taylor in his 1911 essay, "The Principles of Scientific Management." In it, Taylor states that *the most prominent single element in modern scientific management is the task idea* [2]. Taylor encourages organizations to break complex operations down into their elemental tasks, optimize the way in which each task is performed, and provide detailed instructions and plans for the execution of the task. The goals of being faster, better, and cheaper are applied at this task level. According to Taylor, the solution to maximizing productivity is discovering or developing the *one best method* to perform each task [3].

Organizations that implement scientific management also tend to measure productivity at this discrete task level. Key metrics focus on task speed, output, and cost. A problem occurs when an organization correlates these measures of task productivity with value creation.

The Perils of Local Optimization

When the task is made king, all things serve the task. A task focus is inherently myopic. It leads management to create organizational structures best suited to accomplishing the task. Efficiency experts establish the best way to execute the task through a formal process of definition, measurement, analysis, and optimization. Quality experts implement controls to ensure that the process stays within the defined tolerances. Workers are trained to complete the tasks and management evaluates their performance based on how well they comply with the process.

At its root, Lean challenges this fundamental notion of local optimization. Lean measures of success are related to cycle time, financial return, and customer satisfaction [4]. None of these Lean metrics are collected at the task level. Rather, the metrics are considered from a customer-centric perspective and applied to the entire process of getting the customer what they want. For example, cycle time measures the reliable, repeatable, and sustainable time it takes from receipt of a customer request to deployment of the solution in the customer's environment. Financial return is validation of the business case – are customers willing to pay for the results? Customer satisfaction is the ultimate proof of concept. Does the solution support the outcome the customer hoped to achieve? Evaluation of the organization's success with Lean requires foresight in identifying these customer-focused metrics and discipline in collecting and analyzing the data.

Lean in Software Development Has Its Own Challenges

Organizations that produce software are often set up to optimize at the task level. This task-level optimization results in the creation of organizational structures along specialized functions such as requirements engineering, systems architecture and design, programming, quality assurance, and operations. Work progresses in sequential phases with control gates at phase boundaries to ensure that the tasks associated with a given phase have been satisfactorily completed prior to going to the next phase. Tasks in a given phase are performed by the organizational group specializing in the principle function associated with the phase. For example, a requirements engineering group performs the tasks associated with requirements gathering and analysis phase. When a group completes their tasks and fulfills the requirements of the control gate, their work product is handed off to a group that specializes in the function performed in the next phase. Work progresses in sequential phases with control gates at phase boundaries to ensure that the tasks associated with a given phase have been satisfactorily completed.

This sequential approach to software development has been termed a *waterfall*, after a model described by Winston W. Royce in [5]. Ironically, Royce does not use the term *waterfall* to describe the model, and states that a sequential implementation of the model is *risky and invites failure* [5]. However, the model fit well with scientific management's focus on task optimization, and despite Royce's warning, many large commercial and government organizations established sequential waterfall-like development models following Royce's publication. The effects of these waterfall models are still with us today, impacting management approach, organizational structure, process, performance evaluations, and motivational systems.

Treating software development as a project can complicate matters. The Project Management Institute defines a project as a unique endeavor with a defined beginning and end [6]. For most development teams, this end occurs on acceptance and release of the application into a production environment. At this point the development team disbands and team members move on to other projects. Ongoing maintenance and enhancement of the application falls to the operations group. Who, then, is responsible for the whole life of the delivered system? Is it the development group, the operations group, or someone else?

What sequential development methods and hand-offs from development to operations fail to take into account is that customers' needs evolve and change over time. Often, customers do not really know what they need until they see a working product and then they usually need something different from what was delivered. The systems development life cycle does not end at release into production. A working product in the hands of the customer is just the beginning. After all, most of the typical application's life happens after initial deployment.

Short-term trade-offs during pre-

deployment can also have adverse impacts in the long term. For example, quality often loses the optimization battle between scope, cost, schedule, and quality. A dollar saved cutting quality upstream can cost hundreds of dollars or more later.

When looked at from a Lean perspective, the *local* optimizations just described result in decreased efficiency of the overall process serving the ultimate customer (see the "Who's Your Customer" sidebar). For example, if a customer desires a change or new feature, the change must navigate a complex change request management process. Changes disrupt the process and require considerable time to implement. In contrast, Lean sees the delivery process as a stream carrying value to the customer – the *value stream*. The object is to make value *flow* to the customer rapidly and continuously throughout the product's life. To make the transformation to a Lean enterprise, organizations must restructure work and workers to optimize flow and thus reduce cycle time.

Seeing the Value Stream

To see the value stream, organizations must step back and redefine the problem from the customers' viewpoint. Customers do not care about task efficiency. They care about results. The problem is not about task or departmental efficiency or effectiveness. Lean enterprises define the problem in terms of achieving success in the customers' eyes.

Success from the customers' perspective requires that organizations consider the entire value stream from initial concept to delivery through the entire life of a product or service, concluding with the product or service's ultimate retirement. This long-range view encourages global rather than local optimization of the delivery of value to the customer. To maintain continuity in the value stream, the team that creates the product stays with the product throughout its life. There is no handoff from a development team to an operations team: There is just one team serving the customers' needs. In this team's work queue, new features sit alongside maintenance items and are prioritized from a customer perspective with a goal of getting the highest priority items in the hands of the customer as rapidly as possible.

Agile Software Development: An Implementation of Lean Thinking

Recently, the increased popularity of Agile methodologies has led organizations to investigate and, in some cases, adopt iterative and incremental delivery. Agile software development replaces a single monolithic delivery cycle lasting several months or years with a regular cadence of brief delivery cycles lasting no more than a month. Each of these cycles produces a potentially releasable part of the overall application. The aim is to get working features to the customer sooner. Close collaboration with the customer results in delivery of the highest priority features first. The rapid delivery of tangible results permits early inspection and enables course correction when such correction can actually make a difference in getting customers what they truly need. The customer decides when to stop the regular delivery cadence based on whether the value they are realizing from the team's work justifies continued investment.

Unfortunately, many organizations fail to realize the full potential of Agile software development simply because they limit its implementation to the existing technology delivery group. This speeding up of the software development process is just fixing one part of the value stream. At the end of the day, the only thing that counts is working software in the customers' environment – by definition, the end of the stream. Combining Lean with Agile encourages a holistic view, with the customer truly becoming part of a delivery process that spans the full life of the application. Effective Agile teams already know this.

Changing Viewpoints

Successfully transitioning to a Lean enterprise requires that organizations complete the following:

- Identify your ultimate customer.
- Move from a task-centric to a customer-focused perspective.
- Establish goals for cycle time, financial results, and customer satisfaction and measure progress against these goals.
- See the whole value stream.
- Organize around flow.
- Involve the customer.

Lean is not just about doing things differently. Organizations must change the way they see the problem. The first step in this change initiative is to change yourself.◆

Who Is Your Customer?

A few years ago I was conducting a Lean-Agile overview session for a group of workers in a software shop that builds middleware for cable TV set-top boxes. During a discussion about flow of value to the customer, I posed the question, "Who is your ultimate customer?" After a few blank stares and quizzical looks, people began to call out: "My boss?" No. "The cable company?" No. "The developers that use our APIs". No. Finally, a programmer in the first row ventured: "Some dude." I asked him to explain. He went on, "Some dude sitting on a couch watching TV." Right.

References

1. Womack, James P., and Jones, Daniel T. Lean Solutions. Free Press, 2005: 15.
2. Taylor, Frederick Winslow. The Principles of Scientific Management. New York, W.W. Norton & Company, 1967:39.
3. Ibid: 25.
4. Poppendieck, Mary, and Tom Poppendieck. Implementing Lean Software Development. Boston, Pearson Education, Inc. 2007: 238-241.
5. Royce, Winston W. "Managing the Development of Large Software Systems." TRW, Proceedings IEEE WESCON Aug. 1970: 2.
6. A Guide to the Project Management Body of Knowledge. Third Edition, Newtown Square: Project Management Institute, Inc., 2004.

About the Author

Jim York is a Certified Scrum Trainer and co-founder of FoxHedge Ltd. For more than 20 years as a management and IT consultant, he has led, trained, and coached hundreds of individuals, teams, and organizations in the implementation of both Lean and Agile concepts. York's workshops blend his practical experience in Scrum, Lean Software Development, eXtreme Programming, Agile project management, product management, and traditional project management. He shares his passion for Lean and Agile as a frequent presenter at conferences, users groups, public and on-site workshops, and as a business process coach.

FoxHedge Ltd
18899 Maplewood LN
Leesburg, VA 20175
Phone: (703) 771-8367
E-mail: jim.york@mac.com

Coming Events

June 2-5
2008 Homeland Security Science and Technology Stakeholders Conference
Washington, D.C.
www.ndia.org

June 8
PLAS 2008
Programming Languages and Analysis for Security
Tucson, AZ
http://research.ihost.com

June 9-12
2008 Better Software Conference and Expo
Las Vegas, NV
www.sqe.com/BetterSoftwareConf

June 17-18
Enterprise Security Management
Bellevue, WA
www.afei.org

June 23-26
5th Annual Combat ID and Force Tracking
Vienna, VA
www.forcetrackingevent.com

June 23-27
2008 NSMMS
National Space and Missile Materials Symposium
Henderson, NV
www.usasymposium.com

2009
2009 Systems and Software Technology Conference
city, ST
www.sstc-online.org

Coming Events: **Please submit coming events that are of interest to our readers at least 90 days before registration. E-mail announcements to: nicole.kentta@hill.af.mil.**

Cover Comments

Dear CrossTalk Editor,
Hi! Enjoy your magazine! We were wondering about the picture on the cover of February 2008 with the tug pulling the freighter through a canal/gorge. Is the picture real? If so, where was this taken?

—Tim A. Roe
<tim.a.roe@boeing.com>

Publisher's response: I'm amazed by the response February's cover has generated. Our cover artist, Kent Bingham, has given us the following response:

I was forwarded your inquiry about the February 2008 CrossTalk cover. Here's some information on the image: The photo is of the Corinth Canal in Greece, which links the Gulf of Corinth in the northwest with the Sardonic Gulf in the southeast. Here are a couple of links to information about it: <http://en.wikipedia.org/wiki/Corinth_Canal> and <www.grisel.net/corinth_canal.htm>.

I did a few digital manipulations of the original image, but most of what you see is the actual photo (I did reduce the size of the tugboat pulling the larger ship to show a more pronounced contrast between small and big).

—Kent Bingham
<kent.bingham@hill.af.mil>

Dear CrossTalk Editor,
One minor glitch about the cover of the March issue of CrossTalk. In chess, white always moves first, so if a black pawn is being moved, it has to be the second move, not the beginning. Otherwise, a great magazine!

—Stephen J. Chizar
<stephen.chizar@navy.mil>

Publisher's response: We had that same discussion on this end, but decided this is the second move which we still considered the beginning of the game. You're not the first to provide this feedback, so I guess we sparked a little controversy.

Welcoming Software Into the Industrial Fold

James M. Sutton
Lockheed Martin Aeronautics

Software has long been the odd man out in business: It operates in ways that are different than, and often incompatible with, the disciplines of other industries, even when it is teamed with those disciplines under the same enterprise. It generally underperforms other industries on productivity improvement, integration success, quality, and customer satisfaction. Applying Lean production to software enables it to become an industry that works well with classic industries. Lean greatly increases software's contribution to enterprise and customer success.

Software development can greatly improve its business performance by discovering and embracing its kinship to classic (non-software) industries. Perhaps the most important thing software has to gain is guidance on how to implement Lean production.

Lean production as we know it today began in the automotive industry, first with Ford in the early 1900s, then evolving rapidly with Toyota in the 1950s and beyond. Lean is, at heart, a model for maximizing productivity. It turns the assumptions of mass production – the previous model for productivity – on its head. Over the last 50 years, nearly all the classic industries have moved to the Lean model. These industries have, on average, doubled their productivity while tripling their quality [1]. They have also improved their ability to integrate components into systems (integratability) to please their customers.

Software evolved separately from the classic industries and never adopted Lean[1]. Could Lean production, as it is understood and applied in the classic industries, improve software development? First, we show evidence that Lean not only works for software, in some ways it works even better than it does for the classic industries. Then we give an overview of one incarnation of a Lean software process.

What Can Lean Do for Software?

In a nutshell, Lean is about maximizing value and minimizing waste. This means that Lean projects do what matters to business success, and *only* what matters.

Software grapples with value and waste like any other industry. Several software programs at the author's company, Lockheed Martin Aeronautics, have used Lean techniques similar to those practiced in classic industry. The following sections discuss the effects Lean has upon each of the success areas mentioned previously: productivity, integratability, quality, and customer satisfaction.

Productivity

The size of software systems on aerospace programs has grown more than 100 fold since the mid-60s. Early systems like the C-5A weighed in at 50 thousand software lines of code (SLOC) or less. Current systems such as the F-35 are projected to reach six million SLOC or more at delivery. Other domains have grown at least as much, challenging the ability of software development approaches to keep pace.

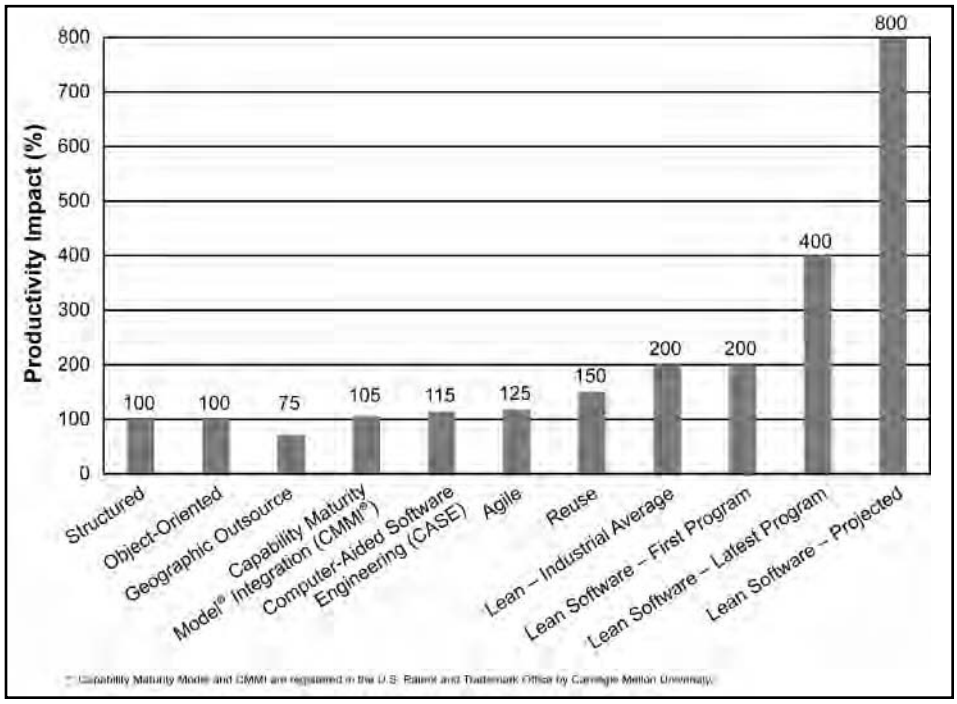
How much has the productivity of traditional software development improved since the early days? The left side of Figure 1 shows results published by proponents [2, 3, 4, 5, 6, 7] of various software approaches, with the structured techniques of the 1970s taken as the beginning reference value. Published claims are accepted without critical scrutiny but are averaged where multiple figures are cited[2]. All these numbers have been rounded up to the nearest multiple of five.

Even the 50 percent productivity gain from reuse, the most impactful non-Lean method listed, does not come close to keeping up with the 10,000 percent plus growth in software system size over the years.

Now to Lean. The industrial average figure for Lean comes from the classic automotive industries via the International Motor Vehicle Project (IMVP) [1]. The Lean software figures come from Lockheed Martin programs that have applied Lean principles to software. Those metrics have been collected and vetted through standard company processes.

The first reasonably full application of Lean to software in the author's experience occurred on the 382J program, a commercial upgrade of the C-130 airlifter [8]. Its productivity improvement matched the average improvement for classic industry due to Lean. Of course, individual businesses in classic industry often exceed a 100 percent average increase. Later Lean software projects exceeded this as well, ending at 400 percent on the latest program, the C-27J. The projected num-

Figure 1: *Productivity Improvements*



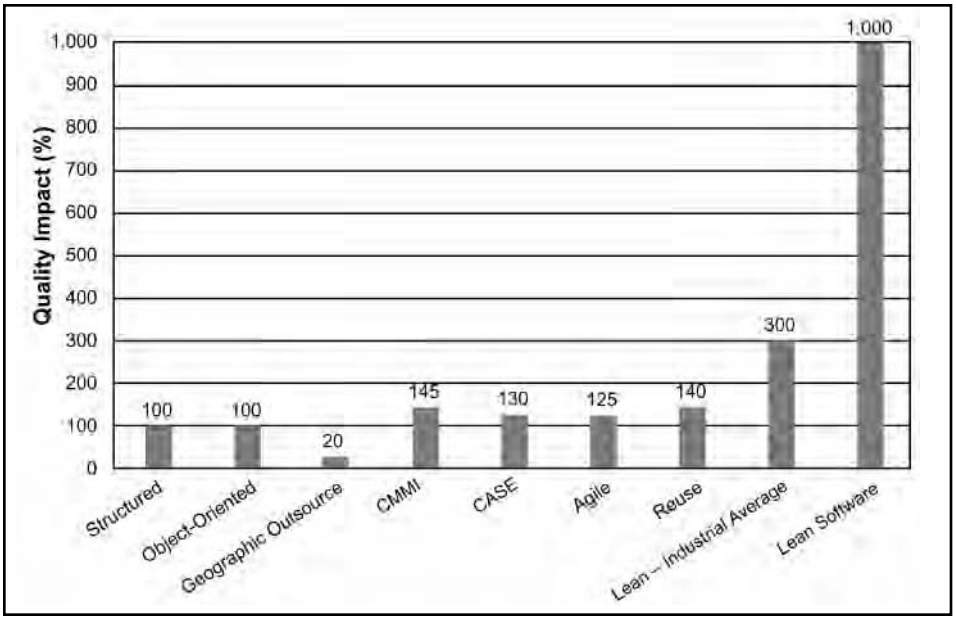


Figure 2: *Quality Improvements*

ber is based on a straightforward extension of process and tools identified by the software team.

Software productivity responds well to application of the Lean principles. Nevertheless, even 800 percent productivity growth is not enough on its own to make 10,000 percent-plus bigger programs successful. For that, major improvements must also be made to integratability.

Integratability

Source-line productivity can be increased by making a concerted push to speed up coding, but any gains are generally at the expense of the build process. In these conditions, errors are more likely to be introduced and less likely to be detected. Downstream builds are less likely to work correctly on the first try. Big projects require that the software coming out of integration be correct both functionally and structurally each time, and be on time. They can neither afford a process full of ongoing repair and reintegration loopbacks nor the cascading delays they bring to the rest of the project.

Programming languages and design structure set the ceiling for software integratability. Most of the widely accepted software languages are better at facilitating rapid source-line production than successful integration. As professor and head of the computer science department at the University of Northern Iowa, Dr. John McCormick recorded what happened when his students formed teams to program and integrate small systems of around 15,000 SLOC:

> During the first six years … students developed their control code in C. No team successfully implemented minimum project requirements. To ease student and teacher frustrations I made an increasing amount of my solutions available to the teams. Even when I provided nearly 60 percent of the project code, no team was successful in implementing the minimum requirements. [9]

The process breakdown occurred in integration. Remarkably similar results have been observed for C's successor, C++. Moving to a language optimized for integration (Ada) finally solved the problem for McCormick's students. It also worked for the Lean software projects mentioned previously (SPARK and Ada). But resistance to such change remains high among software professionals, and with good reason. Rewards and recognition are given for increased productivity in SLOCs (and, occasionally, quality), but almost never for builds that reliably work the first time as expected. Programmers are not to blame; the problem lies with rewards in the current software culture.

Classic industry has been able to use Lean techniques to reduce production cycle times – including integration – by up to 90 percent[3]. Adapting such techniques to the Lean software projects allowed *Lego-Block-like* assembly. Builds were developed and released once per week for several years. Each cycle began with new or modified requirements (an average of five, many of them complex) and ended with verifying that the entire system was still sound. This eliminated the typical *fix 1 SLOC*, *add 1.4 SLOCs of error* experience. With only a handful of exceptions, builds always worked correctly the first time, even though one of the programs, for reasons outside the software process, needed major releases four times as often as usual.

By facilitating significant growth in integratability and productivity, Lean showed a way for software to keep up with the exponential growth in system size experienced to date. Whether software will be able to keep up with future exponential growth remains to be seen, but the Lean projects to date have not even implemented everything in the Lean principles yet.

Quality

Quality, as we use the term, relates to code and supporting-artifact defect densities. Approaches that improve productivity often improve quality even more, as seen in Figure 2.

As before, all numbers for the non-Lean approaches are obtained from their proponents[4] and are reported as published[5].

The Lean software number comes from customer-sponsored independent verification and validation that compared the complete delivered Lean product to the delivered software systems of other vendors on the same aircraft. The results exceed those of classic industry by a considerable margin. It is probably not because this is an exceptional application of Lean: it is, after all, still an early experiment. A more likely explanation is that software is a nearly ideal candidate for Lean. It lacks many of the limitations that apply to material goods (e.g., inherent error tolerances) or services (e.g., soft or human psychological issues).

A metric that Lockheed Martin Aeronautics has used to keep things real is productivity times quality (divided by 100 to retain a percentile scale). One can easily improve productivity at the expense of lowered quality, or improve quality at the expense of lowered productivity. The multiple gives a better idea of how much help Lean – or any other approach for that matter – is actually giving to the software development process. On that measure, Figure 3 shows the software improvement approaches giving the best results, rounded up as before, and in order of increasing effectiveness.

This measure shows that Lean is not only effective for software development; its effects are balanced across both productivity and quality.

Customer Satisfaction

Few metrics have been kept of customer satisfaction in the software field. Evidence

is largely anecdotal, difficult to compare between competing approaches, and subject to challenge. Nevertheless, the Lean software efforts referenced in this article have their share of customer stories.

At the final acceptance board meeting for the 382J software, the United Kingdom acceptance authority said "This is the most nearly perfect software process we've yet seen." The same authority had previously refused to flight-certify several helicopters, which had sat inactive on a tarmac for more than a year because they lacked sufficient evidences of integrity to satisfy his standards.

An early attempt at applying Lean ideas (pre-382J) delivered software to a very picky department head at Tinker Air Force Base in Oklahoma City. This customer was known for returning everything to vendors for rework. The system, called the C-5B Ground-Processing System Software Update Program, was used for five years and no bug was ever reported back to the developers. The department head dubbed the development team "the new breed."

The customers of all the Lean software projects have made positive comments about the software they received. None has given any negative feedback.

A Lean Software Process

As noted earlier, Lean production starts with principles and culture. Lean culture is based on human respect, as well as a general expectation that people will spend most of their time doing things that actively benefit the enterprise or the customer (i.e., adding value). One survey [10] estimated that Toyota employees spend 80 percent of their time adding value, while employees of non-Lean businesses spend just 20 percent. Software is no exception, and there is a lot of room for improvement.

The Lean principles are as follows [11]:

- Value: Being enterprise- and customer-driven.
- Value stream: Preventing erosion of value and eliminating waste throughout the life cycle.
- Flow: Removing discontinuities between development tasks.
- Pull: Starting process control with the customer and passing it *upstream*.
- Perfection: Making it impossible to introduce defects, or else catching them quickly.

The means available for implementing these Lean principles include processes, methods, techniques, and tools. Nevertheless, Lean must actively address culture and management philosophy while

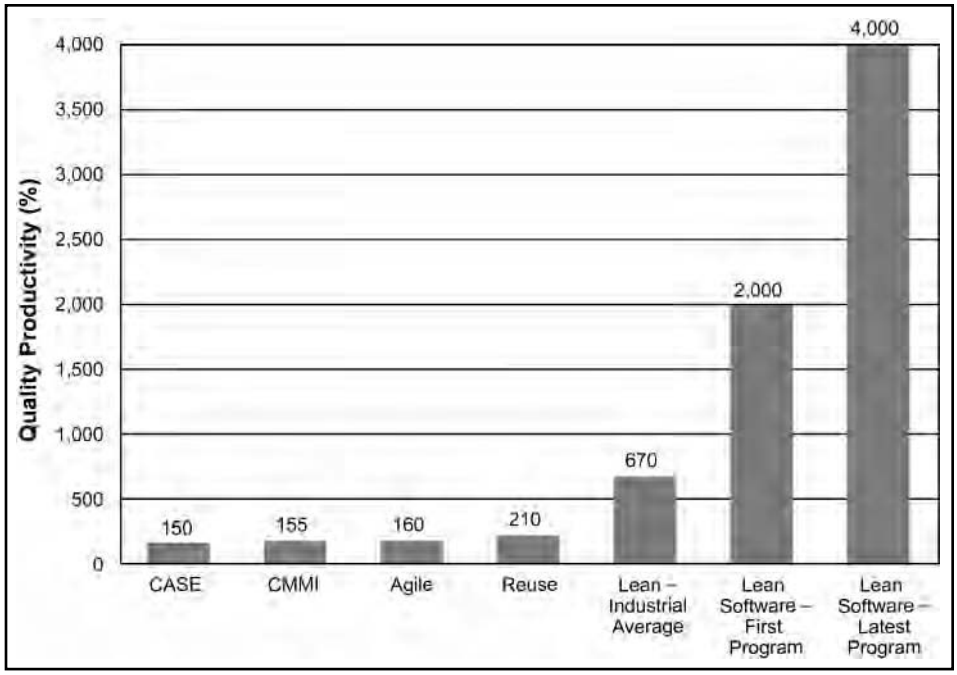


Figure 3: *Productivity Times Quality Improvements*

– and preferably before – it reworks processes, methods, techniques and tools. Otherwise, the Lean improvements will never fulfill their potential and what is gained will erode over time. However, that is a separate topic.

How the Lean principles are implemented in software is discussed in the following section, under the major software lifecycle activities of requirements, design, coding, integration, and verification. For brevity's sake, the techniques cannot be fully explained; however, references are provided.

Requirements

After developing a solid business case and obtaining sufficient project resources, the next most important success factor for software projects is to develop the best possible understanding of what the customer wants. In traditional software development this understanding comes in the form of upfront requirements. Agile projects converge upon it gradually through iterations of product releases and ongoing customer feedback. Both approaches have pluses and minuses.

However, what both traditional and Agile often miss is the customer's motivations. Lean calls these *values*. They include needs, wants, preferences, and responses to problems. Some values may not directly relate to specific aspects of the product at hand, yet could influence what goes into the product. For instance, different cultures process written information in different ways (e.g., right to left, bottom to top, and the significance of different colors). If you target a product to a culture other than your own, your user interface may be workable but not *just right*. They might not even be able to verbalize this; they will just know that they prefer the product of a competitor who understands them better. Identifying these things can be called *value resolution*. It is the foundation of a Lean product life cycle.

Understanding customer motivations will often lead you to requirements the customer might never have come up with on their own; ideas that not even an Agile team would have identified. Thus, value resolution and requirements analysis are different yet complementary as shown in Table 1.

Methods for identifying values include canvassing (literature searches, market

Table 1: *Value Resolution versus Typical Requirements Analysis*

	Value Resolution	Requirements Analysis
Focus	Customer Priorities	Product functionality
Expansiveness	Product line/Future oriented	Single product/Present or past oriented
Main Concerns	Customer delight, business	Organizational "realities"
Detail Level	"What" and "When"	"What," "When," "How"
Other Concerns	Systematic	Sporadic
Customer Involvement	Central	Peripheral

surveys, focus groups), brainstorming, the five why's (asking *why* something is the way it is, then *why* to the answer, and so forth until reaching the root cause), techniques from the Harvard Negotiation Project [12], domain analysis (identifying common and recurring characteristics of the customer's world), and Affinity Diagramming (one of the seven Management and Planning tools that involves grouping lower-level desires or preferences until higher-level values have been determined) [13].

Once you know the customer's values, the next step is to prioritize and, ideally, characterize them. Prioritization can be done with the Analytical Hierarchy Process, a technique for prioritizing a list by pairing all the items in the list, comparing the pairs, and using matrix math to generate a high quality quantitative ranking for each item in the list. This process has been used extensively in must-do situations such as international arms control [14]. Characterization can be done using Kano Modeling, a technique originally developed for the consumer photography market, to identify how customers will react to various potential product characteristics; either *must have*, *delight*, or *proportional* to how well they are done. Kano modeling can also be used on values and requirements [15]. The information these two techniques provide helps the program keep the customer *sold* and engaged, and uses program resources more efficiently.

Requirements are developed from values, and create specific guidance for the particular project. Table 1 gives some idea of the types of information to add (however, minimize negatives like *peripheral customer involvement*). For instance, textbooks state that requirements must omit *how* information. In reality, software development is almost always embedded in a larger development process where requirements must account for constraining higher-level system architectural decisions. There is almost always a certain amount of *how* either stated or assumed in real-world development. Those must be omitted from values, but must be adequately considered in requirements.

Leveraging the Lean Flow and Perfection principles depends on having requirements that possess four important qualities: completeness, correctness, non-redundancy, and unambiguity. The combination of these is sometimes called *requirements integrity*. It minimizes rework backflow (aiding Flow) and prevents many types of errors from entering the process in the first place (implementing Perfection). Two good approaches for achieving these qualities are the Four-Variable Model (FVM), a way of modeling the desired effects of a system on the characteristics of its surrounding environment as functions of the current state of characteristics of that environment [16], and Software Cost Reduction (SCR), a method that applies FVM to the specification of software systems [17]. SCR-style requirements are particularly well suited for embedded systems [18].

Design

Design defines product structures that are optimized for the customer's usage environment. Optimal structure ensures the system will achieve its purposes. It also minimizes the work involved in translating the functionality represented by the requirements into the form needed to go into that structure for implementation. The ideal is one requirement in just one place in the structure (e.g., one method or module in one object). That minimizes the non-value added work done between the productive work of requirement and design. This is the primary goal of Lean Flow. The Lean software projects cited earlier achieved this by developing an architecture that matches the structure of the FVM itself; then, SCR requirements had a natural place to go. This strategy also serves Lean Pull as it becomes easy to add functionality just in time; for instance, as needed to get particular features to the customer in interim releases.

Modern Quality Function Deployment (QFD) [19] is a method that assists with identifying the work that is needed to add value, and provides the means to track that work to assure everything gets done. It typically takes just days to perform and, in return, cuts weeks or months of waste from the software process.

Coding

The Perfection principle says to eliminate the causes of defects before the defects themselves can be introduced. Some programming languages do this better than others. SPARK eliminates all language ambiguities and is optimized for interfacing and for static (i.e., without executing) analysis. Ada has some of these characteristics, and has more options. The Lean software projects used both.

Problematic aspects of C and C++ from a Lean perspective were noted earlier. Their use does not completely negate Lean, but it does leave a lot of unnecessary and removable waste in the development process.

Integration

The Lean Principle most applicable to integration is Flow. Flow removes discontinuities between production steps. Integration seeks to make the code pieces combine immediately and effortlessly, allowing quick progression into verification and delivery.

The most useful Flow technique for software has been Design for Manufacture and Assembly (DFMA). DFMA leads to designing components that are *big parts*, i.e., parts that reduce the total component count but also simplify their integration. In software object-oriented (OO) terms, this means fewer, but generally not bigger classes, covering more of the total system. Standard OO alone will not lead to *big parts*; that requires a suitable abstraction of what software is and does. The concepts involved are mostly alien to software literature and are rather extensive, though not overly complicated so are explained elsewhere [8]. Dr. David Parnas' work was especially useful in developing a software DFMA approach.

Verification

Effective verification is the natural result of applying Lean to all the other activities. Spreading the implementation of requirements across various parts of an architecture, as is typically done in traditional design, forces the verification effort to correctly identify the extent of that spreading and also account for side-effects due to interactions between the pieces and other requirements found in the same design areas. All this work is unnecessary and can be eliminated if requirements have integrity and are isolated within the design.

SCR-style requirements remove the need to develop separate test cases for requirements-based testing. Such requirements are fully suitable test cases in their own right. This eliminates another development effort that is substantial on many programs.

A programming language that permits no ambiguities reduces the need for special verification techniques to detect compiler-based differences and unaccounted for paths (e.g., Federal Aviation Administration, modified condition/decision coverage testing). Such tests may still have to be performed for regulatory or legal purposes, but based on experience with the Lean programs they

will go quickly and uncover few defects.

Conclusion

Lean production has, on average, doubled productivity and tripled quality for the classic industries. Early applications of Lean to software have exceeded those results. Software development is an ideal subject for Lean because its product is pure information that lacks the physical limitations of durable goods as well as most of the soft issues of service activities. In software development, Lean can remain focused on the primary issues of value and waste. This allows the Lean tools to work with unusual effectiveness.

One of the biggest challenges for software development approaches has been to keep up with the growth in size and rigor of customer systems. Lean scales up easily for large systems. It works well in plan-ahead life cycles such as the Department of Defense acquisition system. Lean provides the evidences and assurances needed for safety-critical and high-security applications. These capabilities make it well-suited for the defense and aerospace domains, and for most other domains as well.

It is time for software to take its place as a classic industry and leverage the strengths of Lean production. Lean enables faster code production, smoother integration with other products, fewer surprises to budget and schedule, better quality, and happier customers. Lean converts software from management's biggest worry into one of its best means for assuring business success. Embracing Lean ushers software into the fold of classic industry as a welcome and synergistic partner.◆

References

1. Womack, James P., Daniel T. Jones, and Daniel Roos. The Machine That Changed the World. New York: Free Press, 2007.
2. Potok, Thomas E., Mladen A. Vouk, and Andy Rindos. "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment." Software – Practice and Experience. 1999 Vol. 29, No. 10: 833-847.
3. Ramasubbu, Narayan, and R.K. Balan. Globally Distributed Software Development Project Performance: An Empirical Analysis. Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM. Sept. 2007.
4. Lien, Richard, and Rolf W. Reitzig. Case Study: Realizing 40 Percent Software Development Productivity Improvements Using RUP, Agile, CMM/CMMI, and Organizational Transformation Methods. CMMI Technology Conference and User Group. Denver, CO., Nov. 2004.
5. Tsuda, M., Y. Morioka, and M. Takahashi. "Productivity Analysis of Software Development With an Integrated CASE Tool." Hitachi. 1990.
6. Agile Alliance. "Survey: The State of Agile Development." VersionOne. Aug. 2007.
7. Lim, W.C. "Effects of Reuse on Quality, Productivity, and Economics." IEEE Software Vol. 11, Issue 5. Sept. 1994: 23-30.
8. Middleton, Peter, and James Sutton. Lean Software Strategies: Proven Techniques for Managers and Developers. New York: Productivity Press, 2005.
9. McCormick, John W. "Software Engineering Education: On the Right Track." CROSSTALK. Aug. 2000 <www.stsc.hill.af.mil/crosstalk/2000/08/mccormick.html>.
10. Kennedy, Michael N. Product Development for the Lean Enterprise: Why Toyota's System Is Four Times More Productive and How You Can Implement It. Richmond, VA: Oaklea Press, 2003.
11. Womack, James P., Bruce M. Patton, and Daniel T. Jones. Lean Thinking. New York: Free Press, 2003.
12. Fisher, Roger, and William Ury. Getting to Yes: Negotiating Agreement Without Giving In. New York: Houghton Mifflin, 1992.
13. Brassard, Michael. The Memory Jogger Plus + Featuring the Seven Management and Planning Tools. Methuen, MA: GOAL/QPC, 1996.
14. Saaty, Thomas. Decision Making for Leaders. 3rd ed. Pittsburgh: RWS Publications, 2001.
15. Sauerwein, E., F. Bailom, Kurt Matzler, and Hans H. Hinterhuber. The Kano Model: How to Delight Your Customers. International Working Seminar on Production Economics Innsbruck, Austria. 19-23 Feb. 1996 <www.competence-site.de/dienstleistung.nsf/3397D512929D8241C1256AD8004B0027/$File/kano-model.pdf>.
16. Madey, Jan, and David L. Parnas. "Function Documents for Computer Systems." Science of Computer Programming 25.1 (1995).
17. Kirby, Jim. Rewriting Requirements for Design. Proc. of the International Conference on Software Engineering and Applications. Cambridge: MA. 4-6 Nov. 2002.
18. Alspaugh, Thomas. SCR-Style Requirements <www.ics.uci.edu/~alspaugh/software/SCR.html>.
19. The QFD Institute <www.qfdi.org>.

Notes

1. Lean terminology has occasionally been applied to software ideas, but usually with only a passing resemblance to the meanings used in classic industries. More importantly in each case the author has seen, the underlying framework of thought that ties those words and concepts together in typical Lean Production is missing.
2. Some figures required additional derivation based upon information in the sources. Details are available from the author upon request.
3. According to the IMVP, the Massachusetts Institute of Technology initiative that evaluated Lean adoption in the automotive industry beginning in the 1980s.
4. Generally, the same sources as for productivity. The full explanations are omitted here for space. Details are available from the author upon request.
5. Again, allowing for any derivations needed to present the data in this format.

About the Author

James M. Sutton is a Certified Professional Systems Engineer and a Principal Engineer at Lockheed-Martin (LM) Aeronautics in Fort Worth, Texas. His book on Lean software, "Lean Software Strategies," won the 2007 Shingo Prize. As the lead software product and technical-process architect on several LM programs, he began developing a Lean software approach. Sutton holds a Black Belt in Modern QFD, is a certified Theory of Inventive Problem Solving (TRIZ) practitioner, and has spoken and published for numerous conferences.

101 Academy DR
MZ 8557
Fort Worth, TX 76108-3800
Phone: (817) 935-4011
Fax: (817) 935-5228
E-mail: james.m.sutton @lmco.com

Are the Right People Measuring the Right Things? A Lean Path to Achieving Business Objectives

Paul E. McMahon
PEM Systems

This article provides experiences and guidance applying Lean and Agile techniques together with the Capability Maturity Model® (CMM) Integration (CMMI®) framework to aid measurement and to help in achieving business objectives. While many believe the CMMI produces non-Lean practices, an underlying premise of the article is that both Lean and Agile techniques can be CMMI compliant, and when used together with selective CMMI Level 4 and 5 practices can help organizations achieve business objectives faster. This article goes beyond measurement by addressing the more comprehensive question: Are the right people measuring the right things and taking the right actions at the right time? A case study, six lessons, two process improvement insights related to business objectives, and a commonly held myth about the CMMI framework are included.

The effectiveness of the CMMI framework at helping organizations achieve business objectives is currently a hotly debated topic [1]. One of my clients responded to my recent suggestion to consider using the CMMI and Agile techniques together by stating: *We do not place much value in the CMMI because we have not seen a difference between Level 4 and 5 organizations and those rated lower.* At a recent conference in a presentation about high process-maturity project failures, we also heard some failures were due to *non-process effects such as people* [2]. To help us understand what might be going on, I would like to share a case study.

Case Study Background

I have a client who has been using the CMM/CMMI framework for many years to aid process improvement. According to traditional literature, they appear to be doing the right things.

They have discussed and captured their business objectives using the Goal-Question-Metric technique, and they have aligned their measures with these objectives. They have standardized their processes and trained their people [3]. This client has multiple product lines. Their processes and training include a strong emphasis on product baseline management with disciplined change approvals. Their established and documented business objectives include increasing off-the-shelf product sales, reducing unique customer customization, and meeting cost and schedule commitments.

To help achieve these objectives, the client has been striving to increase management visibility of work through the use of standard product and process metrics. This all seems to make sense. So a natural question arises: Are they achieving their business objectives?

While isolated success stories exist, most senior managers – as well as the company metrics – indicate the organization has fallen far short of its goals.

While the organization continues to propose and bid off-the-shelf solutions, standard metrics show significant unplanned product changes and cost and schedule overruns. A frequent refrain heard at senior management reviews is the following: "Why are we making all these unplanned and un-bid changes?"

So what happened? While this organization appears to have followed the prevailing wisdom in deploying sound processes, why have they failed to achieve their objectives?

This case is not unique. Over the past 10 years, I have observed variants of this pattern in multiple organizations. A closer look at this case may provide insight that can help answer why many organizations seeking higher process maturities are failing to enjoy the promised results of their investments. Let us start with measurement fundamentals.

Measurement Fundamentals

Few, if any, would disagree with the need to measure. But before any organization starts a measurement program, clear objectives and a plan are needed. A fundamental purpose of measurement is to guide management decision making [4]. But how do you manage these measurements to facilitate their effective use?

There is a great deal of literature available today describing the importance of employing a *project management database* as the *basis for retaining process measurements for process management* [5]. Within the Quantitative Project Management (QPM) Process Area of the CMMI, Specific Practice 2.4 identifies the expectation of recording statistical and quality management data in the organization's measurement repository [1].

Another fundamental question faced is deciding what to measure. Watts Humphrey describes three categories of measures: product, process, and resource [3]. Examples of each are:

- Process measure: Defects found by phase responsible.
- Product measure: Defects found by product component.
- Resource measure: Hours per defect fix.

These categories are referred to as *foundation* measures, and it is expected that organizations use these as a *starting point* to derive more meaningful and useful measures *specific* to their business needs [3].

Measurement in Case Study

In our case study, measures in all three of the categories described previously were collected for years, retained in an organizational measurement repository, and the purpose of the measurement program was documented and measurement training deployed.

Independent Analysis and Findings

Due to concerns related to the non-achievement of business objectives, I was asked by my client to conduct an independent analysis. My analysis began with an examination of the data that had been collected for years in the organizational measurement repository. I observed from the *Defects Found By Phase Responsible* data a high percentage of defects being injected late in the development cycle (e.g., test and integration phases). But when I talked to developers, I heard that the majority of their problems were due to vague requirements that did not receive appropriate analysis during the early phases (e.g., requirements and analysis phases).

First Key Observation

My first key observation was this disconnect between the objective data in the organizational measurement repository

built up over multiple years and multiple projects, and what I was hearing from talking to people in the trenches. Rather than selecting the phase that was actually responsible for the defect (e.g., requirements) many people had erroneously selected the current project phase (e.g., integration), causing the organizational repository to be inaccurate.

More Findings

Trying to better understand what was going on, I interviewed a number of workers and asked them to describe how they do their jobs. Prior to these interviews, I had reviewed the organization's standard processes. This was a high-process maturity organization that had previously achieved a formal CMM Level 3 rating.

As the workers described how they did their job, I detected more disconnects between how they said they worked and what was written in their processes.

I asked one developer to describe the product baseline management process. I expected to hear about the disciplined change approval process that was described in their standard processes and training material. But he said, *it's not how we really work*. When I asked the developer to explain further, he said, "We propose things that are similar to what we are going to do, but not exactly. We propose based on where we think the product will be in the future when we think the job will come in. Often those assumptions are wrong."

Second Key Observation

My second key observation was this disconnect between the documented processes and what I was hearing was happening in the trenches.

Stepping Back

I have found the two observations described, to varying degrees, as common patterns in many organizations. Let us explore how organizations get into these situations and how this affects the attainment of business objectives.

As-Is Versus To-Be Process

When process improvement efforts are initiated, there are usually two relevant process views referred to as the *as-is* process and the *to-be* process. The usual approach is to first capture the *as-is*, then discuss weaknesses so appropriate resources can be applied to move the organization toward the desired *to-be*.

If we do not know the as-is then we do not know how big a *stretch* it is for the organization to get to the desired to-be. The hardest part of process improvement is not defining processes, but deploying and teaching those in the trenches appropriate changes in behavior.

Lesson One: The First Step to the Right Measures Is Capturing the Real *As-Is* Process

Often, the *as-is* process does not receive appropriate attention. The common argument goes like this: We are looking at getting better, so does it not make sense to focus on where we want to be? The answer is yes and no.

While it is true that we want to create a clear vision of where we are going, we also need to understand what it takes to get there, which first requires an understanding of where we are. By jumping over the as-is, we sometimes skip *critical dialogue* that helps us understand what and why we do what we do today. This dialogue helps us understand which weaknesses are highest priority and most in need of addressing now. This will, in turn, affect the right things to measure now.

Digging Deeper: Looking For Candidate Root Causes

I wanted to find the root cause of why the organization was not achieving its business objectives. I had heard a number of projects were currently overrunning cost and schedule, so I asked a number of workers, "Does the company underestimate when it bids?" I received a mix of answers.

One project engineer responded, "No, our bids are okay. But we often do not get the hardware ordered and installed in time to meet the software integration schedule."

Another said, "… the bid was okay given the assumptions we made at the time of proposal, but when we find that the assumed baseline product functionality is not there after contract award, we do not adjust the schedule or resources for the additional work we now know we have to do."

Another said, "Yes, we sometimes underestimate because the people who do the bids do not always understand all the pieces of the products they are proposing. After we win, we find out there are impacts that were never planned."

I now had some areas to investigate looking for candidate root causes which included the following:

- Hardware procurement and installation processes.
- Plans and schedule update processes.

But I did not have quantitative data to back up what I was hearing, so I went back to the organization's measurement repository. Unfortunately, when I looked again at the measures being collected, I found it difficult to tie the existing historical data to potential candidate root causes.

This data, which had been collected over many years, were the typical measures found in textbooks (e.g., defects by phase responsible, defects by product component, hours per defect). These traditional measures were not *specific* and context relevant enough to help in identifying and analyzing the root cause candidates.

Lesson Two: Company Standard Metrics are Often Insufficient for Real Process Improvement

In the book, "Measuring the Software Process" the author tells us that when planning for measurement "… experience has shown that it is important to identify the critical factors," and that "… critical factors often arise from concerns, problems or issues that represent levels of risk that threaten your ability to meet your goals … or commitments [5]."

In our case study, cost and schedule commitments were apparently being threatened by late hardware and unplanned, un-bid work. Projects were implementing the standard organizational metrics that had been in place for years, and more context-relevant measures had not been derived. These standard measures were inadequate and could not provide the specific objective data required to address the current bottlenecks in the organization.

Lesson Three: Derive Specific Measures for Needed Insight

Examples of specific measures that could potentially help the organization include:

- Cycle time to get critical path hardware on order.
- Cycle time to install and test critical hardware.
- Counts and cost of unplanned changes to product components.

We derived these measures by asking questions related to the organization's business objectives and what we heard from talking to the people in the trenches such as the following:

- Why are we not getting the hardware installed on time?
- Why do the schedules not reflect all the real work?

How Did This Company Get Into This Situation?

Many of us were taught we need to gather large volumes of data before analyzing and using it. The argument often goes like this: We should not use the data for analysis until we have collected a sufficient amount to ensure it is statistically significant. But the flip side is that the value of data erodes over time. Today's projects use shorter development cycles and environments, technology, tools, and people change fast causing the data to become less relevant faster.

Referring back to the first key observation: Why did the data tell a different story than the people in our case study? The answer to this question is because the people were telling us what was happening today from the context of the projects they were currently working on. The data had been collected over many years, and over many diverse projects and environments, and was not well monitored for accuracy.

In "Understanding Variation," Don Wheeler tells us:

> Much of the managerial data in use today consists of aggregated counts. Such data tends to be virtually useless in identifying the nature of problems ... The work of process improvement requires specific measures and contextual knowledge [6].

Lesson Four: Someone Needs to Care About the Data

In our case study, people had been trained in the importance of the company standard metrics. Nevertheless, with schedule pressures it had become commonplace for the data to be entered quickly and often without adequate consideration for accuracy. Since no one had an immediate reason to care about the data, there was no motivator for people to take the time to ensure they were entering accurate information.

Lesson Five: Use Small Project Teams to *Derive* Meaningful Measures and Review and *Refine* in *Short Cycles*

In the book, "Measuring the Software Process," an example of data collection using the Personal Software Process℠ (PSP℠) is provided [5]. The point is made that *because of a short feedback cycle, the engineers realize the effect of the PSP, and use their own performance data to gauge their improvements*. With PSP, developers save their own performance data and, therefore, have immediate access to it.

One of the reasons PSP works is because the engineer gets that immediate and personal feedback concerning their own behavior. If the data reflected their behavior from years earlier, or reflected another engineer's behavior from a different project, they would be less likely to take action to improve.

The same measurement principles that work for individuals, such as PSP, can be effective for small teams. The closer the data is in time and environment to the current team situation, the greater the likelihood it will be used for real improvement.

"The same measurement principles that work for individuals, such as PSP, can be effective for small teams. The closer the data is in time and environment to the current team situation, the greater the likelihood it will be used for real improvement."

Why Involve People Through Small Teams?

Experience has shown that using small teams works better than large teams because they tend to focus faster and take ownership of real project issues. This leads to identifying the right specific measures and, ultimately, the right process improvements to help an organization meet its objectives faster by addressing key bottlenecks. This is particularly true when the issues faced cross department boundaries, as was the situation in our case study.

This is consistent with Lean principles such as *empower the team* and *eliminate waste* [7]. For related information, refer to Lean manufacturing experiences [8, 9, 10]. This is also consistent with what Agile teams do through their retrospective or reflections workshops [11, 12].

It may be surprising to learn that these techniques also support the recommendations provided through the CMMI guidelines within the Level 4 QPM Process Area. A CMMI guidelines tip states that, "… the specific practices of QPM are best implemented by those who actually execute the project's defined process – not by management or consulting statisticians only." Another tip in the guideline states that, "… when effectively implemented, QPM empowers individuals and teams by enabling them to accurately estimate and make commitments to these estimates with confidence [1]."

This leads to a question: When should an organization consider implementing higher CMMI Level 4 and 5 practices? Before answering this question, let us look closer at the relationship between Agile, Lean, and CMMI Levels 4 and 5.

Agile, Lean, and CMMI Levels 4 and 5

The CMMI guidelines tell us (through a sidebar tip) with respect to Level 5 Causal Analysis and Resolution (CAR), "Although this PA is commonly used for defects, you also can use it for problems such as schedule overruns and inadequate response times that should not be considered defects" [1].

What I find interesting about this tip is the recognition within the CMMI guidelines that defects and problems may both be addressed through common practices. This may alarm some traditionalists, but a primary focus of Agile is value to the customer rather than being overly concerned with categorizing work as a defect or a requirement. This view is also consistent with *Lean thinking* where a defect is defined to be anything that does not meet the *customer needs* [13].

Lean Six Sigma has evolved from the two initiatives Six Sigma and Lean. The focus of Six Sigma has been on reducing defects and reducing cycle time through measurement [14]. This is also a key focus of CMMI Levels 4 and 5 practices. The methods known as *Lean* focus on improving *process flow and speed* [13]. While the roots of Agile methods are proven small-team techniques, they have also drawn heavily from Lean manufacturing experiences [11, 12].

One argument I have heard by Lean and Agile proponents against the CMMI is *why do I need to wait until Level 5 to analyze and fix problems?* This is in reference to where CAR is placed in the staged representation of the CMMI [1]. The answer: You do not, and you should not.

℠ Personal Software Process and PSP are service marks of Carnegie Mellon University.

Returning to Our Case Study

In our case study, we addressed two real problems: Cost/schedule overruns, and late hardware. Both had a direct impact on the customer. We needed specific data to verify we were tackling the correct root cause. As soon as the data was gathered, we needed to analyze it and act on it. Fundamental to Lean thinking is continuously identifying the next critical bottleneck, analyzing it, and then taking action to remove it as rapidly as possible [8, 9].

Experience has shown it is on the *analysis* and *action* side where traditional process improvement efforts often fall short. CMMI Level 2 and 3 practices are of fundamental importance, and a level of competency must be achieved in these practices before taking on any Level 4 and 5 practices. But too often, organizations get overly focused on Level 2 and 3 practices to the exclusion of valuable *business-focused* efforts that can be well supported through the higher level practices. Also, as seen in our case study, many organizations that employ only the Level 2 Measurement and Analysis Process Area to drive their measurement initiatives tend to focus on collecting and storing company standard metrics rather than on the critical *analysis*, *communication*, and *actions* to improve.

Process Improvement Insights Related to Business Objectives

Alistair Cockburn has observed a commonality between engineering and manufacturing. You can observe it, "… once you notice *decisions* as the product that moves through a network of *people*" [15]. This observation can be taken a step further to help us gain process improvement insights related to business objectives.

Returning to the case study where the hardware is late causing the software integration schedule to slip, whenever I have investigated similar problems, often the solution comes down to two possibilities:

- Weakness in process.
- People assigned lack skills or training.

The CMMI framework at Levels 4 and 5 provides two similar categories of causes [1]:

1. Common causes of variation (weakness in process).
2. Assignable causes of variation (process not followed).

It turns out in our case study that most of the time the hardware was ordered and installed on time. But sometimes special circumstances occur which perturbs the normal flow of work requiring a *person* to make a *decision*. Sometimes the people closest to the situation could make that decision, but sometimes (often due to inexperience) they decide to defer their decision impacting the normal cycle time. Often some variant of this situation is at the root of such a problem.

In our case study, we found that sometimes the hardware was not ordered on time due to missing data on a procurement requisition and an inexperienced procurement specialist who didn't know how to handle the situation. We also found that sometimes projects were under-bid because product impacts were not fully identified by assigned personnel. We also found that sometimes proposal assumptions proved incorrect and plans and schedules were not appropriately updated.

At this point a question arises: Are these *process* problems (common causes), or *people* problems (assignable)?

In my experience and in this case study, the answer most often turns out to be a mix of both – the process is never perfect, nor are the people.

- **Insight One:** When using Lean and Agile techniques, the distinction between causes of variation matters less than solving the problem immediately and meeting the customer needs.
- **Insight Two:** Besides solving the problem immediately, we need to take appropriate action to minimize the likelihood of the problem reoccurring. Most often this action includes both improving the process or environment and providing additional mentoring and/or training to the people.

Lesson Six: Consider Using Selective CMMI Level 4 and 5 Practices Early Together With Agile and Lean Techniques to Address Key Objectives

One reason we distinguish between types of variation at CMMI Level 4 and 5 is to help us take the right actions at the right time. At Level 4, we quantitatively manage sub-processes and at Level 5, we analyze data and take corrective action. However, this doesn't mean you can't do both at the same time. This approach is supported by the CMMI Continuous Representation. In the case study, the right answer was to provide immediate on-the-job mentoring to help workers now and – at the same time – refine the process to help others in the *future*.

A key value in the Level 4 and 5 practices of QPM and CAR is their potential to help an organization focus on critical *sub-processes related to business objectives* [1]. These focused efforts allow us to gain critical quantitative data leading to timely actions which, in turn, help achieve business objectives faster.

This same line of reasoning can be employed to help process improvement groups *win the battle of the budget* for continued organizational investment in the higher CMMI level practices [16]. Too often after receiving a Level 3 rating, organizational investment in process improvement dries up. This is unfortunate when the greatest business value may lie just ahead.

A Commonly Held Myth About the CMMI Framework

It is well known that Agile proponents *value individuals and interactions over processes and tools* [12]. It is also well known that often when using the CMMI framework, there is a tendency to try to separate *process issues* from people issues [2].

The belief that *people issues* fall outside the CMMI is a myth that rests at the heart of why customers do not see a difference between higher maturity level organizations as well as why many companies are failing to achieve their business objectives when using this model.

This myth is partially rooted in a misunderstanding (and misapplication) of the Generic Practices. I have heard the comment that *the Generic Practices are the same for each process area so there is nothing specific we need to do.* This is incorrect.

As an example, Generic Practices 2.3, 2.4, and 2.5 relate to the activities of providing resources, assigning responsibilities, and providing training [1]. The training of *people* expected by the model is specific to each process area. Therefore, *people issues* fall inside, not outside, the CMMI framework. But keep in mind that nothing says this training cannot be *Lean training* such as *on-the-job* and *just-in-time*.

An Example of the Right People Taking the Right Actions at the Right Time

On a recent Standard CMMI Assessment Method for Process Improvement A (SCAMPI A) appraisal in which I participated, a question was raised with respect to the adequacy of the *organization's measurement repository* which was distributed rather than centralized. When questioned, one of the developers commented, *it works better for us because we carry the measures forward.* His point was that the focus in the company (which uses both Agile and Lean techniques) was less on archiving data in a

centralized repository, and more on *project personnel* analyzing the current data and *taking action* to carry the measures forward, providing *timely* process improvements to help the current active projects.

Agile practices, such as incremental planning, continuous measurement, and retrospective [11, 12], along with Lean practices of eliminating waste, amplifying learning, and delivering as fast as possible [7], provide proven ways that can help an organization achieve its business objectives faster. These techniques can also comply with CMMI practices including the capture of measurements, causal analysis, and taking action to improve [1] – while avoiding the pitfalls observed in our case study.

Conclusion

Ask yourself this: Is your customer seeing the results of your process improvement efforts? If not, do you understand your real *as-is* process? These are the first questions to ask that will lead to measuring the right things.

If you use the CMMI framework, you can also gain the benefits of Lean and Agile techniques. Involve and listen to your people in the trenches to help find the right things to measure leading to timely improvement actions. Lean and Agile techniques are not only compatible with the CMMI framework, they can facilitate your CMMI implementation and help you achieve your business objectives faster.

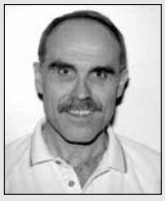
Many high-process maturity organizations today are integrating Agile and Lean techniques into their CMMI-compliant processes. If your organization isn't moving in this direction you may soon find yourself trailing the competition.◆

References

1. Chrissis, Mary Beth, Mike Konrad, and Sandy Shrum. CMMI Guidelines for Process Integration and Product Improvement. 2nd ed. Addison-Wesley Professional, 2006.
2. Hefner, Rick. "How High Process Maturity Projects Fail." Proc. from the 2006 Software and Systems Technology Conference, Salt Lake City, UT.
3. Humphrey, Watts. A Discipline for Software Engineering. Addison-Wesley, 1995.
4. Clements, Paul. Software Product Lines. Addison-Wesley, 2002.
5. Florac, Carleton. Measuring the Software Process. Pearson Education, 1999.
6. Wheeler, Don. Understanding Variation. 2nd ed. 2000.
7. Anderson, Callison. Challenges Faced By Military Programs and Lean Software Development. Proc. of Systems and Software Technology Conference, Tampa, FL, 2007.
8. Goldratt, Eli. The Goal. 3rd ed, Northriver Press, 2004.
9. Kennedy, Michael. Product Development for the Lean Enterprise. Oaklea Press, 2003.
10. Liker, Jeffrey. The Toyota Way. McGraw-Hill, 2004.
11. Schwaber, Ken. Agile Project Management With Scrum. Microsoft Press, 2004.
12. Cockburn, Alistair. Agile Software Development. Addison-Wesley, 2006.
13. George, Mike, David Rowlands, and Bill Kastle. What Is Lean Six Sigma? McGraw-Hill, 2004.
14. Pande, Holpp. What Is Six Sigma? McGraw-Hill, 2002.
15. Cockburn, Alistair. "What Engineering Has In Common With Manufacturing and Why It Matters." CROSSTALK, Apr. 2007.
16. Reifer, Don. "Profiles of Level 5 CMMI Organizations." CROSSTALK, Jan. 2007.

Did this article pique your interest?
You can hear more at the 2008 Systems and Software Technology Conference April 29-May 2, 2008, in Las Vegas, NV. Paul E. McMahon will present in Track 5 on Tuesday, April 2, at 2:40 p.m.

About the Author



Paul E. McMahon, principal of PEM Systems, helps large and small organizations as they move toward increased agility. He has taught software engineering, conducted workshops on engineering process and management, published articles on Agile software development, and authored "Virtual Project Management: Software Solutions for Today and the Future." McMahon is a frequent speaker at industry conferences, and is a certified ScrumMaster. He has more than 25 years of engineering and management experience working for companies, including Hughes and Lockheed Martin.

PEM Systems
118 Matthews ST
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org

Letters to the Editor

Dear CrossTalk Editor,

I agree with many of the sentiments expressed in the article, *Computer Science Education: Where Are the Software Engineers of Tomorrow?* (January 2008), but I disagree with the authors' singling out of Java as the cause of the problem. Yes, computer science students should be learning more formal methodology and more math. Yes, they would benefit a lot from learning several different programming languages. Yes, learning a low-level language like C or Assembler would help give students a better understanding of what's going on at the machine level when a computer program executes, and this knowledge would be beneficial to professional programmers.

The article makes a good point that the first language a programmer learns will influence the way they think about problems later. This is true, and I agree that teaching programming with a scripting language is a bad idea, but it seems like Java is a perfectly good language in which to learn programming. It has a beautiful core language structure with most of the features one would want from a modern programming language: packaging, class inheritance, exceptions, separation of interfaces from implementations, and so on. A student can learn programming with Java at a superficial level just using existing classes to quickly get something working, or the student can go much deeper. It seems like how easy or hard a programming assignment is depends on the curriculum and the professor more than it depends on the programming language the student is using.

Personally, Fortran was my first programming language, but I long ago learned to write programs that are *not* like Fortran. Honest.

The authors say it's a bad thing that, when programmers use a big class library or package library, often they have no idea what is going on *under the covers*. The authors are proponents of using Ada. A core goal of Ada is that a program should be built with a set of well-documented interfaces (package specifications in the case of Ada) which encapsulate functionality and can be used as a kind of *black box* to do useful things. The programmer who is using a package has to understand from its specification what it is supposed to do, but in most cases does not need to understand the underlying implementation. I remember when I first went to a presentation on Ada by language designer Jean Ichbiah, abstraction and information hiding were the aspects of the language that he emphasized.

It seems to me that the Java class libraries achieve this goal better than Ada (at least better than Ada 83 or Ada 95, which are the versions of Ada with which I am familiar). Large-class libraries come standard with Java. The class libraries encapsulate a huge amount of external functionality: networking, graphics, input/output, math, and hundreds of other areas. Most of the source code is available to view when necessary. There is so much more available in these classes than in the corresponding standard Ada packages that there is almost no comparison between the two languages in this area, and in most cases the Java class library abstractions are really well designed.

Yes, it can be a bad thing if a programmer misuses a built-in class or package, or uses it as a way to avoid understanding something he should understand to be competent in the core domain area that his program is trying to solve. But mostly the existence of lots of built-in classes is a good thing. We don't want everybody (or every company or university) reinventing their own version of standard functionality. We want a base upon which programmers can build, using their creativity to create new things.

It is different for students. The authors are right that industry programmers will mostly be using predefined packages (Ada) or classes (Java). They don't need to know the internal details of how the network works, or the graphics processor, or the compiler, or the operating system, or the database. On the other hand, students should be learning these things – especially computer science majors. But please don't blame Java.

—Mitchell Gart
<mitchell.gart@kronos.com>

Dear CrossTalk Editor,

This issue (February 2008) is an excellent choice of articles! Lots of us face these issues all the time. Thanks much.

—Chuck Lundquist
<chuck.lundquist@usaa.com>

Dear CrossTalk Editor,

It is interesting to me that all the articles in the February 2008 issue have titles and discussions that are really too generic. I have a lot of trouble getting my Department of Defense (DoD) customers to accept the front-end investment in DoD Architecture Framework (DoDAF) and Capability Maturity Model Integration (CMMI) processes. They expect to get to coding as soon as possible and I know why – that is not my problem. My problem is that when professionals do not bluntly articulate how to apply, for example, processes built for large programs to small projects, they may get published but that does not help. What I want to know, as a project manager and devotee, is precisely how did the small project reviewed actually modify DoDAF and CMMI guidance to effect modern development methods and get acceptance of the modified project processes? By the way, at 10,000 feet I know what small versus large is, but when I get up close to that elephant, what is defined as small? Metric budget? Lines of code? Complexity of integration, of functionality, of task decomposition, operational decomposition, communications, safety criticality . . . ? Wait, don't tell me: The answer is yes! In operational systems, what exactly changes in the testing, quality assurance, and verification and validation processes between large and small efforts? Can the Nevada Test Site folks clearly articulate this?

—Andrew S. Loebl
<loeblas@comcast.net>

Publisher's response: I am sorry you did not get the information that you were looking for. I have the same concerns when reviewing articles and I thought this had been addressed in the article, "Why Do I Need All That Process? I'm Only a Small Project." *In the online version of their article, the authors include links to the actual three processes discussed in their article and the resulting condensed process. Few organizations are willing to share such information because of its sensitivity and I was grateful for their openness. With the hope that perhaps you missed these examples, I'll include them here:* <www.stsc.hill.af.mil/crosstalk/2008/02EP%20 Online%20Access%20eBG-PD-22-Schedule%20Management.doc>, <www.stsc.hill.af.mil/crosstalk/2008/02/EP%20Online%20Access%20eBG-PD-21-Scope%20Management%20Crosstalk.doc>, <www.stsc.hill.af.mil/crosstalk/2008/02/EP%20Online%20Access%20eBG-PD-31-Resource%20Allocation.doc>, and <www.stsc.hill.af.mil/crosstalk/2008/02/EP%20Online%20Access%20ESP-PD-33-Scope%20Schedule%20Resource%20 Management.doc>.

Measuring Continuous Integration Capability

Bas Vodde
Odd-e

Continuous integration (CI) is an important Lean software development practice. Measuring the capability of your CI environment provides a road map for improvement and an aid for sharing practices between projects. The CI grid, introduced in this article, is a simple, question-based metric for checking the current CI capability of a project.

For the last few years, Lean thinking has been applied to software development, introducing Lean software development [1-3]. The roots of Lean thinking are in the Toyota Production System (TPS) [4]. The TPS has two pillars: autonomation (Jidoka) and just-in-time. Autonomation is also described as *automation with a human touch* and is based on the idea that machines automatically stop when a mistake is detected so that it can be fixed immediately and no material, effort, or electricity is wasted. A CI system brings autonomation to Lean software development.

Software integration is often a problematic area in product development. CI is a common technique used to overcome these traditional problems. CI means that developers integrate their software as frequently as possible (at least daily), in small steps (small batches), to prevent sudden surprises. Increasing the integration frequency requires making it easier to integrate, and often means reducing processes such as formal inspections and approvals. (Peer reviews and personal code reviews are certainly still valuable.) Different mechanisms need to be in place to ensure the quality of the integrations. This is where a CI system provides the support – the safety net – that enables CI. A CI system always compiles the software and runs all the tests [5]. When one step fails, the system stops like an autonomated system and will inform the person who likely broke it. Such a capability is essential in modern Lean software development. Important questions should include: What is the capability to continuously integrate in the project? How about other projects in the company?

This article introduces a grid for making the CI capability visible. This can be used for planning improvements and sharing practices.

In 2005, the grid was introduced in Nokia Networks, making telecommunications equipment, with the goal of measuring the current CI capabilities in the teams moving to Agile development. In the two years that we have used the grid, it has provided a target and visibility of improvement in the area of CI.

History

Daily building was made popular by Microsoft in the '90s and cited as a best practice in the book "Rapid Development" [6-8]. Extreme Programming, introduced in the late '90s, took daily building to the extreme and introduced the concept of CI – check-in in small steps, then compile and test everything during each check-in [9, 10]. The introduction of tools such as CruiseControl [11] makes setting up a CI environment easier and is making CI more popular.

> *"Software integration is often a problematic area in product development. CI is a common technique used to overcome these traditional problems. CI means that developers integrate their software as frequently as possible ... to prevent sudden surprises."*

CI Grid

Background

The ultimate goal of CI is to always have a shippable, working product. Some features might not be implemented completely, but they will not break the product.

In the CI grid, we assume two levels of automated integration and testing. The first level is CI. A build in CI is triggered by a very short time-based trigger (e.g., five minutes) or by a check-in in the Software Configuration Management (SCM) system. The system is then compiled and tested. Due to the time it takes to execute all tests in a large project, it is not useful (or possible) to run all the tests. In the first level – CI – the focus is on providing quick feedback. The second level is daily builds. Daily builds are executed nightly. A daily build has a slower feedback cycle with the result being ready before the morning. Thus, in daily builds, the automated test set can be much larger and ideally contains all automated tests. The shorter CI cycle prevents the daily build from breaking frequently.

Especially in large product development, the CI and daily build can happen on both whole-product level and on subsystem level. However, only having CI and daily build on the subsystem level causes integration problems and does not create the ability to have a shippable, working version of the product every day. The feedback time for CI is essential and therefore it might be needed to have the CI on subsystem level. When having CI on subsystem level, the daily build can stay on product level and can catch subsystem integration problems early.

The described environment is definitively not the only possible way of implementing CI. Two levels might not be the best solution for very large projects and is too much for smaller projects. However, this environment offers a good starting point for most projects. If tests can be run easily all the time, then one level might be enough. If there is trouble running all tests in the daily build, then focus should be placed on speeding up the build and tests. Projects rarely need more than two levels.

CI Grid Overview

The CI grid is a tool for making CI capability visible in either an organiza-

tion or in a single project. The benefits of making this visible are:

- Give guidance for projects to improve their capability.
- Share practices between different projects. Projects can learn from each other.
- Give improvements a higher priority. (What gets measured gets done)

One warning upfront: The grid only measures the environment and capability to use CI. CI is a practice – a habit – of the development team. The grid does not measure if developers are integrating their code frequently.

The grid is a matrix with questions and metrics. Questions are answered and a color is filled into the table. There are four different possible colors:

- Red (white): Not started.
- Yellow (grey): We are working on this.
- Green (black): Yes, we have this.
- Blue (X): We have no interest in this.

Colors, not numbers, are used for getting a quick overview. (For this article, I'll use grayscale since the print is black and white; in real use, I recommend color.)

The metrics are not intended to be very precise, they can be estimated. The questions are categorized into three groups. Each group contains questions about how well a project is going in:

- Daily build.
- CI.
- Test-driven development (TDD) [12].

Daily Build Questions

The questions in the daily build section are:

1. Compilation

a. Is the whole product compiled (and linked) every day at a fixed time automatically?

If the product consists of subsystems or sub-components then the *whole product* here would mean all of these. *Automatically* means that no manual intervention is needed to start the daily build and that the build itself does not require human attention either, regardless of whether the build succeeds or fails.

2. Sanity check

a. Are essential tests run to ensure the stability of the build?

Essential tests ensure that the main functionality of the build is working. Having just the essential test requires less amount of test automation than the other questions and, thus, can be done fairly easy for projects which do not have much test automation.

3. Unit tests

a. Are all unit tests executed every day after the build compilation?
b. Are the unit tests that developers put in the SCM system automatically included in the build without extra effort from the developers?

In a daily build, it should be possible to execute all unit tests. It must be easy for developers to add unit tests to the build. This is normally done by the developer putting their tests in the SCM system.

4. Installation

a. Is the system installed to *production-like environment* every day after the build compilation?

A *production-like environment* is the environment where a finished product should be installed. This is the hardware and environment it is running. This does not mean that it goes live automatically every day. The installation should also be completely automated.

5. Acceptance tests

a. Are all possible acceptance tests (e.g., functional/system) executed?
b. Can people easily (without much effort) add new acceptance tests?

Some acceptance tests will be running in the *production-like environment*. It is possible that some acceptance tests cannot be executed daily (e.g., because they take too long) – they will be excluded. The acceptance tests are also added to the build in a similar manner as unit tests.

6. Reporting

a. Is a failure automatically reported to the people who might have broken the build and to others?
b. Is the current status being published?

Reporting of the daily build should also be automated. A common mistake is to just mail everyone. This leads to the situation in which people ignore a failed build (because it is not their fault). Thus, reporting should be done automatically to the people who potentially broke the build (anyone who changed something since the last time it was working) and also to other interested parties such as testing or management.

7. Policy

a. Does a broken build (including tests) become the first priority for the project?

If a daily build fails then this needs to be the first priority for everybody in the project. If this is not true, then daily builds will start failing and become completely useless since they do not provide the visibility and stability in the project anymore.

CI Questions

The questions in the CI section are:

1. Anytime integration

a. Can, at any time, any developer integrate his work into the main branch without too much effort?

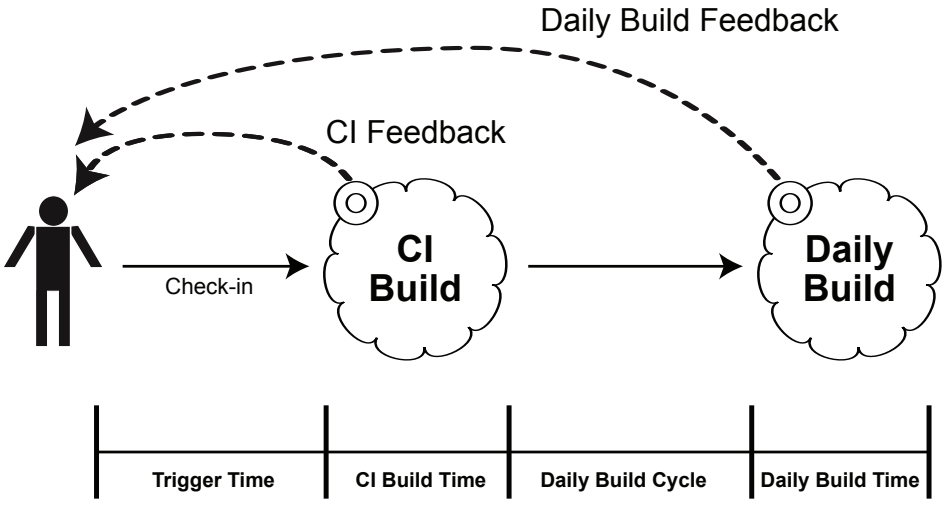
Integrate means that the check-in triggers a compile and test. The *main branch* means that it is integrated into the total product source, not on a separate feature branch.

This question is best understood by looking at what it does not mean. When there are specific integration points (e.g., three-week builds) or when developers work for weeks on subsystems or features without integrating it into the main branch, then there is not an *anytime integration* capability (the grid needs to be filled in with red).

2. Compilation

a. Within an hour after check-in, is the system automatically compiled?

Figure 1: *CI Environment*



The *anytime integration* should trigger a compilation of the software. This works on either a time-based trigger or on a trigger from the SCM system.

The time to feedback is critical in CI and, thus, in larger projects this compilation might be done incrementally or even at the subcomponent level only so that the feedback comes quickly enough.

3. Sanity check

a. Are essential tests run after the system is compiled?

The sanity check for CI is likely to be different than the one from the daily build. This sanity check consists of mainly unit tests and some acceptance tests. It would be best if the build system could find the most relevant tests automatically and execute them. Again, fast feedback is critical and the sanity check must not take too long.

4. Reporting

a. Is a failure automatically being reported to the person who integrated the work?

b. Is the current status being published?

The number of people to which a failure is reported should be smaller than with the daily build. The current status should be highly visible to the team since developers base their integration decisions on the current build status [13]. A lava lamp (using a red and green lava lamp to show the build status [14]) or a public monitor is a good solution for achieving this visibility.

5. Policy

a. Does a broken build become the first priority in the project? (This is the same as for daily build.)

TDD Question

The question in the TDD section is:

1. Are developers doing TDD?

This question is not directly related to CI but it is still included in the grid. Using TDD, a developer's private workspace always stays in a working state. When a developer increases his integration frequency, then he needs to learn to work in smaller steps. This is where TDD makes CI easier. Between the TDD cycles, a developer checks the status of the current build and check-in when the build passes. When the build fails, the developer does a few more TDD cycles and then checks in.

Metrics

The metrics used in the grid are best explained with a picture of the CI environment (see Figure 1 on page 23).

1. Integration feedback time

The integration feedback time is the time between when a developer is ready for a check-in and a CI build report. If a project is not doing *anytime integration*, then this is equal to the build feedback time. Otherwise, this is equal to the trigger time plus the maximum compile and test time.

For example, when a project uses a 10-minute trigger time, the compilation lasts five minutes, and the tests take 15 minutes, then the integration time metric would be 30 minutes. In a normal situation, a developer will know after 30 minutes if the integration has failed or succeeded.

2. Build feedback time

The build feedback time measures the build cycle. When a project has daily builds then this metric should always be one day. When a project has CI and not a daily build level then it is less than one day and the metric is not important anymore. For example, when a project makes one complete build every three weeks then the build feedback time is three weeks.

3. Test coverage

The last metric in the grid is the test coverage (during the daily build). This metric tells something about the validity of all the other fields in the grid since most of the questions do not make sense if the test coverage of the automated tests is low.

Reviews

When using CI, the effort the developer spends for integrating code needs to be minimized. The more effort that is needed for the developer, the less likely he or she is to integrate continuously, and the more likely he is going to save his work up and integrate in a bigger batch. This will seem more efficient to him but counterproductive to CI.

Minimizing the effort before integrating code often means changing reviewing practices. Formal inspections are too heavy to do when integrating multiple times a day. A quick peer-review and personal reviews might work better. Another alternative is to delay the reviewing until after the integration has been done. This way the developers can review all changes done – for example, once a day or once a week. This is easy to plan and the reviewing will be done as a shared effort which also increases the team learning.

4. Grid Example

Table 1 is an example of the grid filled in. This example contains four different projects (P1-P4). The grid starts by listing the technology and platform for each project. The build environment is often dependent on platform and technology and, thus, listing them

Table 1: *Example CI Grid*

	P1	P2	P3	P4
Technology (Program Language)	J, C++	J	J	C
Platform	Lin	Lin	Lin	Hpux
Daily Build				
Compilation				
Sanity check				
Unit testing				
Installation				
Acceptance tests				
Reporting				
Policy				
Continuous Integration				
Anytime integration				
Compilation				
Sanity check				
Reporting				
Policy				
Test-Driven Development				
Metrics (Estimations)				
Integration feedback time	24h	15m	24h	3w
Build feedback time	24h	15m	24h	3w
Test coverage	?	50-80	80	?

Legend

Not started
Working on it
We have it

makes it easier to see which projects you can share practices with.

P4 is a legacy project which does not have a daily build at all; its build cycle is three weeks. Both P1 and P3 have implemented daily compilation but neither is able to execute tests automatically yet. Therefore, their integration feedback time and build feedback time is equal. P2 is a fairly small project and they chose to only have one cycle (no daily build since everything within the CI cycle had been built). However, they only compile and execute unit tests; they do not yet have automated installation and acceptance tests.

From the grid, each project can see their potential improvement areas: P4 could start implementing daily builds, P1 and P3 could implement automated tests, and P2 could focus on automated installation. Also, from the grid we can see that the people from P3 might want to talk to the people from P2 since their environments are similar and they can learn from each other.

Conclusion

The grid offers a very simple way of measuring the CI capability of a project. It can be completed in 10 minutes and it then can provide a road map for improvement and a comparison between projects. The grid, however, limits itself to measuring the capability of the environment and not the usage of that environment by the people in the project.◆

References

1. Womack, J., and D. Jones. Lean Thinking. 2nd ed. Free Press, 2003.
2. Poppendieck, M., and T Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley, 2003.
3. Poppendieck, M., and T. Poppendieck. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2006.
4. Ohno, T. Toyota Production System: Beyond Large-Scale Production. Productivity Press, 1988.
5. Duvall, P., S. Matyas, and A. Glover. CI: Improving Software Quality and Reducing Risk. Addison-Wesley, 2007.
6. Cusumano, M., and R. Shelby. Microsoft Secrets. 1995.
7. McCarty, J. Dynamics of Software Development. Microsoft Press, 1995.
8. McConnell, S. Rapid Development. Microsoft Press, 1996.
9. Beck, K. Extreme Programming Explained. Addison-Wesley, 1999.
10. Fowler, M. <www.martinfowler.com/articles/continuousintegration.html>.
11. "CruiseControl." Cruise Control <http://cruisecontrol.sourceforge.net>.
12. Beck, K. Test-Driven Development. Addison-Wesley, 2003.
13. Fredrick, J. "Continuous Integration." Better Software Magazine Sept. 2004.
14. Clark, M. Pragmatic Project Automation. The Pragmatic Programmers, 2004.

About the Author

Bas Vodde is the owner of Odd-e, a small consulting company based in Singapore, that specializes in training and coaching related to Agile and Lean development. His main interests are in Scrum and especially how to use it within large companies and large projects, and also focuses on technical practices, especially TDD (including refactoring) and CI. Vodde believes you need a well-factored code base if you want to be fast and flexible. Originally from Holland, Vodde moved to China where he worked for Nokia and gained experience on large projects and the traditional ways they are run. After this, he became convinced that Agile is the way forward for all sizes of projects and moved to Helsinki, Finland to introduce Agile and watched teams adopt both Scrum and Agile practices. His interests include Lean Production, quality management, and programming, and he recently co-authored the book "Large Agile and Lean Product Development."

Odd-e Ltd. Pte
Singapore
E-mail: basv@odd-e.com

Web Sites

Principles of Lean Thinking

www.poppendieck.com/papers/LeanThinking.pdf

In the 1980s, a massive paradigm shift hit production factories throughout the U.S. and Europe. Mass production and scientific management techniques from the early 1900s were questioned as Japanese manufacturing companies demonstrated that 'just-in-time' was a better and more useful paradigm. The widely adopted Japanese manufacturing concepts came to be known as Lean production.

Wikipedia's Take on Lean Manufacturing

http://en.wikipedia.org/wiki/Lean_manufacturing

For many, Lean is the set of Toyota Production System *tools* that assist in the identification and steady elimination of waste (Japanese term: *muda*), the improvement of quality, and reduction of production time and cost. The Japanese terms from Toyota are quite strongly represented in Lean. To solve the problem of waste, Lean manufacturing has several 'tools' at its disposal. These include continuous process improvement (Japanese term: *kaizen*), the "5 Whys" and mistake-proofing (Japanese term: *poka-yoke*). In this way, Lean manufacturing can be seen as taking a similar approach to other popular software improvement methodologies.

Lean In the Product Development Process

www.manufacturing.net/lean-in-the-product-development.aspx?terms

As global competition continues to intensify across industries, companies are actively pursuing strategies that will enable them to improve corporate financial performance. Over the past decade, Lean strategies have emerged as the predominant operational improvement strategy, and companies are aggressively pursuing Lean to achieve a competitive advantage. Today, Lean strategies are increasingly being extended to include processes and groups outside of operations – most notably product development. Eliminating risk in execution as new products are brought to market to fuel growth is an imperative to achieving success. However, very few companies have successfully implemented Lean in the product development process. Product development and engineering organizations have, for the most part, only received training in Lean concepts and tools without clear applicability. This site shows the areas that are traditionally weak in Lean, and provides suggestions on how to strengthen business strategies.

Software Adds Muscle to Lean Engineering

http://machinedesign.com/ContentItem/58785/SoftwareaddsmuscletoLeanEngineering.aspx

Lean manufacturing gets lots of press – and for good reason. It drives waste out of manufacturing facilities and makes them more profitable. It is relatively easy to see waste in manufacturing because you can touch it. But it is not so obvious in software engineering departments. Still, Lean manufacturing principles apply to both the manufacturing and software engineering departments as well. In fact, engineering departments are good places to start Lean manufacturing programs.

Win in the Flat World: Apply Lean Principles Across the IT Organization

www.mbtmag.com/article/CA6498453.html

Most adherents see their Lean implementations as something "only useful for the production floor." The author sees Lean as a company-wide tool. Across an organization, Lean practices can be used in engineering, in the front and back offices, in research and development, and even in information technology to improve operations by driving waste from existing environments.

Frequently Asked Questions About Lean

www.mamtc.com/lean/intro

Answers to many Lean questions, including: Can the workings of a "Lean organization" that demands quick, cost-effective adaptability be compatible with the methodical, disciplined processes required by the International Organization for Standardization 9001? There is also a comprehensive comparison between Lean and traditional methods, a compelling essay on the benefits of Lean, and a glossary of Lean terms.

Jazoon.com – Where Java People Meet

www.softdevarticles.com/modules/weblinks/viewcat.php?cid=78&sortid=2

Dr. Alistair Cockburn's article on software engineering is more like manufacturing than most people expect. Once we spot the similarities between the two, we can apply the lessons learned over the last 50 years in manufacturing to software development. This article picks six lessons to apply to software development gleaned from the manufacturing industry. Jens Norin's article focuses on the increasing popularity of Agile development methods and how Agile is putting new demands on the traditional configuration management (CM) discipline. A working CM environment is essential for the rapid nature of Agile development methods, but the CM process and the CM role has to be adapted to the present evolution of software development methods as well as automated tools. This article discusses Lean principles and Agile values within a CM scope, and also introduces a method to classify the CM discipline in relation to development methods and levels of tool automation.

The Lean Nature of Google's Software Practices by Manageability.org

www.manageability.org/blog/stuff/google-development-practice-lean-production

The question posed is "How close does Google's development practices match Lean software development?" In addition, what does Google do that goes beyond Lean software development? On this site, the reader can view each Lean principle and see what Lean has to say about it. Also included are comments on the positive aspects and uses of Lean, including eliminating waste, amplifying learning, team empowerment, delivering the product as fast as possible, seeing the whole picture in terms of software and process, and building in integrity to your product.

Using Both Incremental and Iterative Development

Dr. Alistair Cockburn
Humans and Technology

Incremental development is distinctly different from iterative development in its purpose and also from its management implications. Teams get into trouble by doing one and not the other, or by trying to manage them the same way. This article illustrates their differences and how to use them together.

Incremental and iterative development predate the Agile movement; I first ran into them while doing research for the IBM Consulting Group in 1991 [1, 2, 3]. At that time I learned how different they are in purpose and nature, and eventually how to manage them.

Those differences seem to have been forgotten in the intervening years. I now see would-be Agile teams suffering from doing only incremental development, where I used to see waterfall-type projects suffering from doing neither or only iterative development.

Both are needed. People need to learn to use them separately as well as together.

Definitions, Please!

Briefly,

- **Incremental** development is a staging and scheduling strategy in which various parts of the system are developed at different times or rates and integrated as they are completed.

The alternative strategy to incremental development is to develop the entire system with a *big-bang* integration at the end.

- **Iterative** development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system.

The alternative strategy to iterative development is to plan to get everything right the first time.

It is important to notice that neither strategy presupposes, requires, or implies the other. It is possible to do either alone, both, or neither.

In practice, it is advisable to do both in different quantities. If you only increment, there tends to be an unpleasant surprise at the end when the quality is not good enough. If you iterate the entire system, ripple effects of the changes easily get out of control.

It Is Not Waterfall

First of all, we need to get past the *it looks like waterfall* trap.

In all development, whether prototype, Agile, tornado, or waterfall, we first decide what to build; we then design and program something. Only after doing some programming (however much we decide to program), we put ourselves in a position to debug the system. Only after the system is running can we validate that what we built is the right thing to build, built correctly. This sequence is shown in Figure 1.

Figure 1 should really be read backwards, as a dependency diagram: We cannot ship till we debug and validate; we cannot debug until we code; we cannot code until we design, we cannot design until we have decided what to design.

In other words, the *validation V* is a simple fact of life, and we shall have to deal with it in both incremental and iterative development.

Incremental Development

In incremental development, we break up the work into smaller pieces and schedule them to be developed over time and integrated as they are completed. Figures 2-4 illustrate this procedure.

Imagine that the top sheet of blocks represents various user interface components, the middle sheet represents middleware, and the bottom sheet represents back end or database components.

Figure 1: *The Validation V Is a Fact of Life*

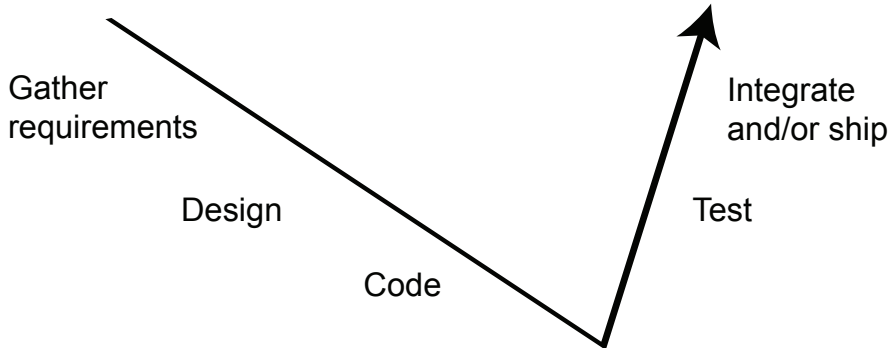


Figure 2: *Incremental Development, Stage1*

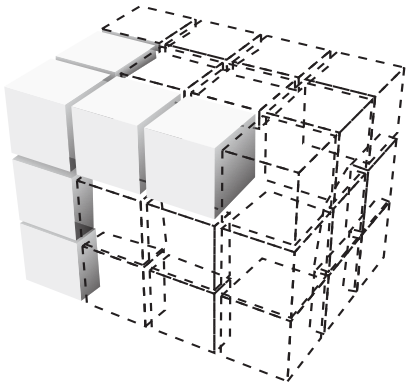


Figure 3: *Incremental Development, Stage 2*

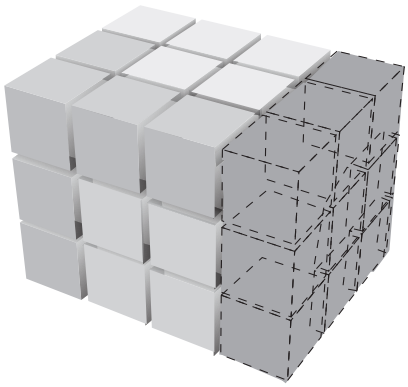


Figure 4: *Incremental Development, Stage 3*

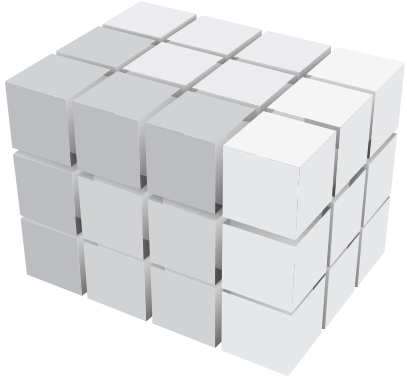


Figure 2 shows that in the first increment, a full piece of functionality is built from the user interface (UI) through to back end (and in this case, additional pieces of the UI are also built). In the second increment (Figure 3), we see that additional functionality is added across all layers of the system. This may be a sufficient point to deploy the system as it is so far to real users and start accruing business benefit. In the third increment (Figure 4), the rest of the system is completed and incremented.

The pattern just described is the one used by most modern projects, Agile or not. It is a staging strategy with a long history of success.

The mistake people make these days is they forget to iterate. They do not factor in the time to learn what they misunderstood when they decided what to build at the very beginning and what needs to be improved in the design.

This mistake results in the old failure of delivering what people do not want. I wish to highlight that even many Agile project teams make this mistake.

The correcting strategy is iterative development.

Iterative Development

In iterative development, we set aside time to improve what we have.

Requirements and user interfaces are the most notorious places where we historically have had to revise our work, but they are not the only ones. Technology, architecture, and algorithms are also likely to need inspection and revision. Performance underload is often wrongly guessed in the early stages of design, requiring a major architectural revision.

In terms of the validation V, the difference is that instead of integrating and perhaps shipping the software at the end of the cycle, we *examine* it from various standpoints: Was it the right thing to develop? Do the users like the way it works? Does it work fast enough?

Figure 5 shows the validation V for an iterative development cycle.

There are two particular, specialized rework strategies:

- Develop the system as well as possible in the thinking that if it is done sufficiently well, the changes will be relatively minor and can be incorporated quickly.
- Develop the least amount possible before sending out for evaluation, in the thinking that less work will be wasted when the new information arrives.

There are aficionados of both approaches. Indeed, both work well under certain circumstances. A project manager must learn to use both.

"Requirements and user interfaces are the most notorious places where we historically have had to revise our work, but they are not the only ones."

The following is an effective use of the first strategy: A musician and a photographer were making a DVD together. The musician recorded the four-minute track in its entirety. The photographer noticed that a small sequence of slide transitions did not match the music. The musician re-recorded just those bars, and spliced them into the music track.

To show an effective use of the second strategy, I adjust Jeff Patton's example of the painting of the Mona Lisa, imagining the discussion between Leonardo and his patron (Figures 6-8) [4].

Leonardo draws a sketch of what he intends to do (Figure 6) and goes to the patron, asking, "How's this going to work for you?"

The patron says, "No, no, no. She can't be looking right, she has to be looking left!" Fortunately, Leonardo has not done too much work yet, so this is easy to change.

Leonardo goes away, reverses the picture and does some color and detail (Figure 7). He goes back to the patron: "By cost, I'm about one-third done. What do you think now?"

The patron says, "No, you can't make her head look that big! Make it look more balanced with her body size." (Yes, they had the equivalent of Photoshop and airbrushing back then – he was called *Leonardo*).

Leonardo goes away and finishes the painting (Figure 8) and turns in his bill.

The patron says, "Really, I'd rather have her eyes bigger, but okay, for the money I've paid, let's call it done."

What I wish to highlight is that both strategies are valid and both fit the *iterative* tag. In both cases, rework was done on an existing part of the system.

Blending the Two

Incremental and iterative development fit well with each other. Taking advantage of the validation V, we can arrange to alternate *incremental* Vs with *iterative* Vs in various ways to get any number of composite iterative/incremental strategies as Figure 9 illustrates.

Figure 9 shows a strategy in which each incremental section of the system is given two examination/rework periods before being integrated and staged for delivery. The figure shows three incremental development periods, each increment staged as it gets completed, the whole then shipped as a package.

This is one – but only one – of the possible ways to blend the two. As long as they are clearly marked *examine* and *stage* (or even better, *ship*), the Vs can be mixed in almost any combination.

Figure 9 shows one added benefit of describing increments and iterations with Vs: The resulting diagram maps easily to a calendar. Each examine, integrate, or ship marker is a milestone in the project manager's project plan. This allows the project manager to preview and monitor the time spent revising. In this way, we put the validation V *fact of life* to good use in showing our incremental-iterative development strategy.

Managing Them

Superficially, the two look very similar.

Figure 5: *The Validation V for an Iterative Cycle*

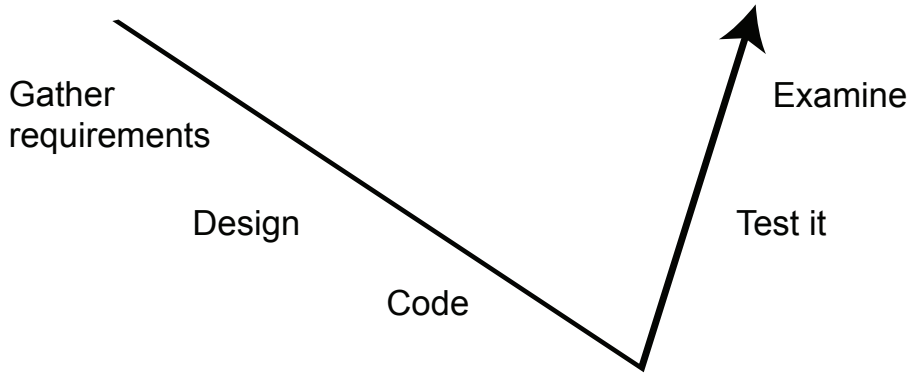


Figure 6: *Iterative Development of the Mona Lisa, Stage 1*

Figure 7: *Iterative Development of the Mona Lisa, Stage 2*

Figure 8: *Iterative Development of the Mona Lisa, Stage 3*

However, they need to be managed differently. Increments are easy to spot, easy to separate, relatively easy to estimate, and easy to schedule. The entire strategy can be summarized in two steps:

- Divide the system into complete, useful slices of functionality or according to some other useful decomposition you may choose.
- Do them one after the other.

Iterations are considerably more difficult. They are hard to separate, hard to estimate, and hard to schedule. Of course, being difficult does not mean you do not have to do all those things. You still have to do them, including answering the following three questions:

- Which items need to get rework periods scheduled?
- How many rework periods does each need?
- How long should each rework period be?

Although there is no simple, reliable answer to these questions, there is a simple starting strategy from which you can derive a version that fits your project [5].

- Plan to revise the user interface for sure, plan time to add or revise requirements as the end users start to use the system, and suspect that the performance-under-load architecture will need to be revised.
- Allocate two revision periods for the user interface design, and one each for requirements drift and performance re-architecture.
- Allocate for the first revision one-third of the initial development time, and for the second revision one half of that.

You will need to develop and test your own numbers, but those might not be bad starting numbers for the first estimate.

If you are doing Agile development with Scrum or eXtreme Programming, make sure every user story or backlog card passes through the Scrum sprint backlog three times with those multipliers on them as work size estimates for the subsequent passes.

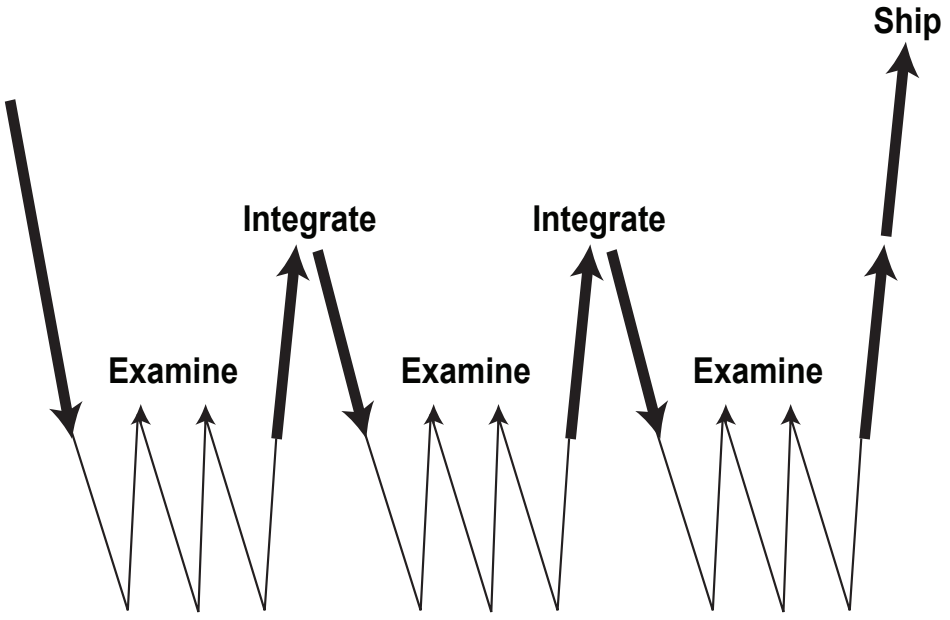
Three Stories

Finally, I offer three stories of incremental/iterative development done poorly and done well. The first one, project *Baker*, shows iterative confused with incremental – they iterated when they should have incremented. The second one, project *Laddie*, shows the problem of modern Agile projects incrementing without iterating. The final one, project *Winifred*, shows them done well.

Project Baker was a fixed price, fixed-scope project with 200 people. They worked in monthlong cycles (an incremental development strategy).

The teams were separated and worked in pipeline fashion so that the requirements writers wrote requirements for some features for a month then passed

Figure 9: *Putting Iterative and Incremental Development Together*



them along at the start of the next month to the designers. A month later, the designers passed the designs along to the programmers who programmed for a month. At the end, the testers were given pieces of code to test and integrate.

Misunderstanding the term *iterative development*, they then gave everyone instructions that requirements and design could be changed at any time (this was their iterative development strategy).

The pandemonium that ensued is just what you might expect. Each month the requirements writers revised any part of the requirements document, which changed any amount of design and programming. By the third month, it was obvious to the programmers that they were programming a system that had already been changed by the designers who were simultaneously aware that they were designing a system that had already been changed by the requirements writers. The testers never got anything that fit together.

Project Baker was in trouble from the start, partly due to the pipelining strategy but even more to the uncontrolled iteration.

Let us look at an Agile failure mode.

Project Laddie was using an Agile approach with two-week iterations. All user stories were put into a long list and developed to completion each iteration (their incremental development strategy). At the end of each iteration, the customer was shown what had been built during those two weeks. Of course, since two weeks is a very short time, there was never time to show the customer what was being designed so there generally were corrections to be made.

The customer had to choose whether to delay work on new user stories in order to correct the mistakes made, or to push the corrections to the back of the work queue. (This was their iterative development strategy – not a very nice one from the customer's perspective.)

The customer complained after a while that he felt he had to get things right the first time since the choices given to him about how and when to fix mistakes were not very pleasant. This, he correctly felt, violated the very spirit of Agile development [6].

Let us end with a happy story.

Project Winifred was a fixed price, fixed-scope, fixed-time project of 18 months, using about 45 people at its peak. The basic development cycle was three months, resulting in deployment after each cycle [7]. (This was the incremental strategy.)

There was no particular incremental strategy required within each development cycle – the teams got to develop features in any sequence they wanted. However, every team had to show their ongoing work to real users at least twice within each cycle, so that the users could change or correct what was being built (their iterative strategy). It had to be real software, from user interface to database, not just screen mock-ups.

Typically, each team showed the users what they were building after six weeks of work and again after eight weeks of work. In the first user viewing, perhaps 60-80 percent of the functionality was complete. The users were given the right to change anything they did not like about what they saw, including *I know that's what I said I wanted, but now that I see it, it is actually not what I want at all.*

By the second viewing, perhaps 90-95 percent of the functionality was complete, and the users were only allowed to correct egregious mistakes and make fine-tuning corrections. This permitted both requirements and user interface correction while still making sense for a fixed-price contract.

Project Winifred deployed successfully and the users got more-or-less what they wanted. The system is still in use and being maintained a decade later, which is a fair indicator of success.

Note that project Winifred's incremental-iterative strategy combination followed the style of the Mona Lisa story earlier.

Summary

The word *increment* fundamentally means *add onto*.

The word *iterate* fundamentally means *re-do*.

Sadly, *iterative development* has come to mean either incremental or iterative, indiscriminately. That was an unfortunate turn for our industry since each serves a different purpose and needs to be managed differently.

Incremental development gives you opportunities to improve your development process, as well as adjust the requirements to the changing world.

Iterative development helps you improve your product quality. Yes, it is rework, and yes, you probably need to do some rework to make your product shine.

The development process, feature set, and product quality all need constant improvement. Use an *incremental* strategy, with reflection, to improve the first two. Use an *iterative* strategy, with reflection, to improve the third.◆

References

1. Cockburn, A. "The Impact of Object Orientation on Application Development." IBM Systems Journal Nov. 1993.
2. Cockburn, A. "Unraveling Incremental Development." Alistair Cockburn <http://alistair.cockburn.us/index.php/Unraveling_incremental_development>.
3. Cockburn, A. "Using VW Staging to Clarify Spiral Development." Alistair Cockburn <http://alistair.cockburn.us/index.php/Using _VW_staging _to_clarify_spiral_development>.
4. Patton, J. "The Neglected Practice of Iteration." StickyMinds <www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=13178>.
5. Cockburn, A. "Three Cards for User Rights." Alistair Cockburn <http://alistair.cockburn.us/index.php/Three_cards_for_user_rights>.
6. Cockburn, A. "Are Iterations Hazardous to Your Project?" Alistair Cockburn <http://alistair.cockburn.us/index.php/Are_iterations_hazardous_to_your_project?>.
7. Cockburn, A. Surviving Object-Oriented Projects. Addison-Wesley, 1998.

About the Author

Alistair Cockburn, Ph.D., is an expert on object-oriented (OO) design, software development methodologies, use cases, and project management. He is the author of *Agile Software Development*, *Writing Effective Use Cases*, and *Surviving OO Projects* and was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group, served as special advisor to the Central Bank of Norway, and has worked for companies in several countries. Many of his materials are available online at <http://alistair.cockburn.us>.

1814 East Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
E-mail: acockburn@aol.com

Sounds Like Quality to Me

While enjoying the view of the bay behind the Tampa Convention Center during the Systems and Software Technology Conference (SSTC) last year, I heard a series of deep melodic tones coming from someplace off to the right. Wondering if we shared the center with a musically inclined organization, I followed them to Platt Street where I found the "music" was created by cars passing over the grate of a bridge. Speed determined the pitch and duration of tone while the combination of notes resulted from the spacing of vehicles. The effect was as if someone were plucking cords on an enormous bass viol[1].

When I received my July edition of CROSSTALK, I was intrigued by Gary Petersen's BACKTALK [1] article based on Gene Weingarten's *Washington Post* article, "Pearls Before Breakfast." These articles describe an experiment in which only two people out of a thousand paid any significant attention to Joshua Bell, one of the nation's best classical musicians, playing his $3 million Stradivarius at the L'Enfant Plaza Metrorail Station in Washington, D.C.

Considering why I could hear music from automobile tires when other people failed to recognize a world-class performance in front of them led me to speculate why I can also see a recipe for disaster in activities that others might consider "cost effective."

Prior to the SSTC, I attended my 35th alumni reunion, including participation in the 100th anniversary of the Miami University Men's Glee Club. So I've developed a pretty good "ear" through singing over the years. Having spent nearly as much time witnessing the results of less enlightened software development practices, I'm also confident that I can recognize what is likely to work and what will not, even without the help of mathematical tools. As Dr. David Cook, another contributor to this column, phrased it, "When you work in quality, you see the world a little differently.[2]"

As for the other 998 people on the Metrorail, I can think of three points that might explain their difference in seeing:

- Experience indicates that great musicians are only heard in concert halls after paying big bucks for tickets. Thus, if somebody is playing in public, he or she must not be world class. Hmm, where have I heard that before? "Of course he doesn't know what he's talking about, he's from quality assurance. If he were really any good, he'd be a developer!" On the other hand, if you are a consultant, your advice is automatically presumed to be at least as valuable as your fee[3].
- Most people entering a Metrorail station are in a hurry to get someplace. I'd be a lot less worried about my retirement if I had a dollar for every time I've heard some variation of "I don't have time to (learn-prepare-implement) a (plan-system-methodology). This (anything) is needed (indication of immediacy)!"
- Some people simply don't appreciate music. This is unfortunate if it is due to a hearing problem, but on the other hand many people just don't care. A manager who isn't interested in quality development processes may attempt to implement a software application by doing nothing more than pointing at a developer and saying, "do it." Occasionally 'it' gets 'done' and these successes are hard to argue with. Just remember that even with odds of 146,107,962-to-1, somebody does periodically win the Powerball lottery[4].

A Point-of-View Gun[5] is the obvious solution to all of this. Unfortunately, in addition to it being entirely fictional, the Code of Federal Regulations "Trespass to Land Owned and Leased by the U.S. Government" (10 CFR 860) prohibits the use of dangerous weapons on government property. Maybe a new television series would help people see things differently? You know, one in which an invisible violin player sneaks into different government software projects each week and then saves the day, the project, the budget, a pretty girl, and the world in general by playing haunting etudes at appropriately dramatic moments. Just remember to put me down for a recurring role as the equally invisible quality assurance specialist.

—Robert K. Smith
rkensmith@earthlink.net

Reference

1. Petersen, Gary A. "Net-Centric Virtuosity." CROSSTALK July, 2008.

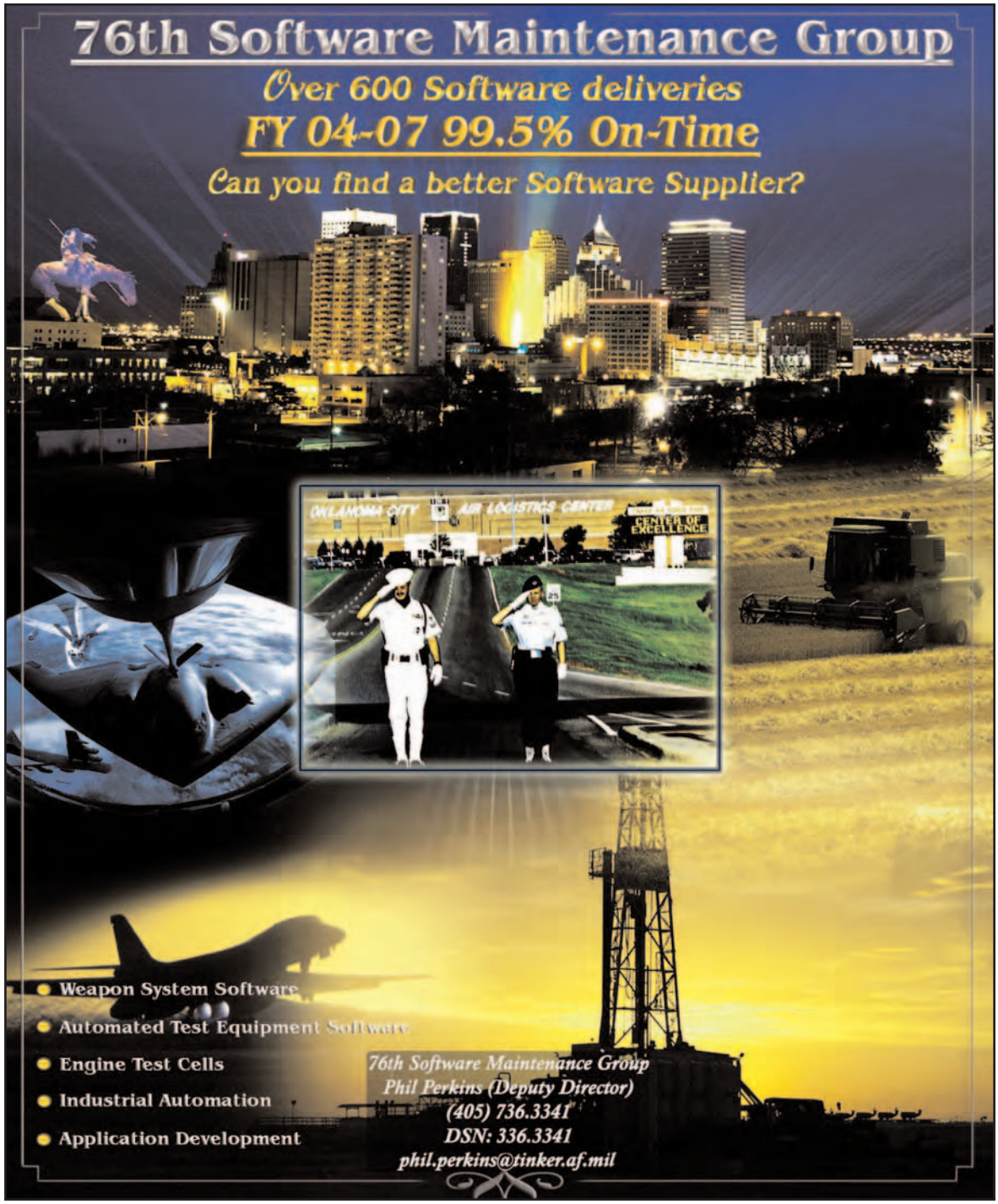
Notes

1. Yes, a bass viol. A viol is a bowed string instrument. Similar to the cello, the viol, or viola da gamba, is played between the legs (hence the name "viola da gamba" or, literally, "leg-viol.") While it is not a direct ancestor of the violin, there is some kinship between the two instrument families. For more information, see <www.vdgusa.org> and <www.wikipedia.org/wiki/bass_viol>.
2. Idle conversation with Dr. Cook in a midwestern airport a couple of years ago.
3. Anyone interested in paying me a lot of money for advice is welcome to contact me!
4. As a visual reference, 146,107,962 grains of rice piled in four-by-three foot layers of 50 pound bags would be a stack more than 12 feet high.
5. Hitchhiker's Guide to the Galaxy – the movie, by Douglas Adams.

Can You BACKTALK?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CROSSTALK, we also accept articles for the BACKTALK column. BACKTALK articles should provide a concise, clever, humorous, and insightful perspective on the software engineering profession or industry or a portion of it. Your BACKTALK article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BACKTALK article, visit our Web site at <www.stsc.hill.af.mil>.



CrossTalk / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737