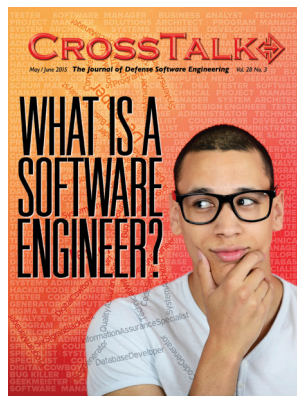


May / June 2015 **The Journal of Defense Software Engineering** Vol. 28 No. 3[illegible]

Departments

- 3 From the Sponsor
- 38 Upcoming Events
- 39 BackTalk



Cover Design by
Kent Bingham

What is a Software Engineer?

- 4 Software Engineering and the Persistent Pursuit of Software Quality**
Software engineering continually seeks to achieve software quality, as the field keeps pace with the changing role of software.
by **Paul D. Nielsen**
- 10 Enough Software Processes, Let's Do Patterns!**
A more effective approach when implementing defined processes is to focus on patterns.
by **Paul E. McMahon**
- 15 The Impact of Contextual Factors on the Security of Code**
Non-technical decisions made in policy, acquisition, governance, resources, processes, and every other aspect of managing software have a direct impact on the resulting operational security.
by **Dan Shoemaker, Ph.D. and Carol Woody, Ph.D.**
- 20 Open Forum: What is a Software Engineer?**
Software Engineering is exponentially more complex than cobbling code together which satisfies a functional requirement.
by **Al Kaniss**
- 22 Is Software Engineering Really Engineering?**
Software engineering may be the prototype of a new form of engineering dealing in the realm of logic as it is applied to functionality to benefit mankind.
by **Lawrence Peters**
- 25 SISE: A Novel Approach to Individual Effort Estimation**
The SISE estimation model presents a novel alternative to PSP's PROBE by combining expert judgment with empirical data to create reasonably accurate predictions.
by **Russell Thackston and David Umphress**
- 31 The Effectiveness Formula**
The Effectiveness Formula is based on three development attributes of success: communication, management effectiveness, and technology.
by **Dr. Randall W. Jensen**
- 36 Open Forum: Meeting Cancelled!**
The old proverb, as originally recorded in Benjamin Franklin's Farmer's Almanac, states that "a rotten apple spoils the barrel."
by **Akbar Khan**

CROSSTALK

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
Crosstalk Online www.crosstalkonline.org

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

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.

Subscriptions: Visit www.crosstalkonline.org/subscribe to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

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.crosstalkonline.org/submission-guidelines. **CROSSTALK** does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

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

Trademarks and Endorsements: **CROSSTALK** 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:
For questions or concerns about crosstalkonline.org web content or functionality contact the **CROSSTALK** webmaster at 801-417-3000 or webmaster@luminpublishing.com.

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

CROSSTALK is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing luminpublishing.com. ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CROSSTALK would like to thank NAVAIR for sponsoring this issue.



What is a Software Engineer?

Software engineers, computer engineers, computer scientists, programmers: how often do you hear people using these titles interchangeably, as if these roles all perform the same job? While, yes, they all do work with computers, that would be the extent of these roles' similarities. Saying these roles are the same is like

lumping master chefs, executive chefs, sous chefs, and short-order cooks together because they can all cook. Software engineers have a very unique set of skills that are quite different from the rest of those roles that 'work with computers.'

Let's start by talking about programmers. I think of programmers from the days where software houses were organized into groups of designers and programmers. When a designer finished their work on a product, they would hand over the design to a programmer, who would then code that design into a specific computer language. This working relationship inferred that programmers were only good at writing software but not designing software. With today's powerful development environments, along with the ever-evolving vernacular of the software industry, programming in this day and age means a lot more than just "slinging" code.

But let's take that old-school definition of the programmer and combine it with the ability to design and inject a heavy dose of computational theory. Now just add a white lab coat and you've got yourself a computer scientist (that's my title - although I have never owned a white lab coat). Computer scientists bring their strength to the table in their ability to model something that needs to be studied in a virtual environment. This comes in really handy when the problem you are studying involves explosives or other things that are hard to analyze in the real world (e.g., river currents, blast fragments, forces from a thrust vectoring nozzle). In my career with the DoD, I spent over a dozen years working on a war-game simulation that the U.S. Navy used to develop military tactics. That war-game simulation contained many models interacting with each other to present a complex environment to the users. For those models to be

able to coordinate with each other, an architecture needed to be established to define and organize the lines of communication. Now we are getting seriously beyond "slinging" code.

As more functionality was added to the simulation, performance bottle-necks began to appear. Now the software team found that they had exceeded the limitations of the hardware they are running their simulation on. "It's not our fault!" they said, "Our system is just what it needs to be. We need faster hardware." Something about 'try a bigger hammer' comes to mind. Enter the computer engineer: this individual would look at the system being used to run the simulation and state: "Of course you are having performance problems, your hardware system is built the wrong way and doesn't support the needs of your simulation." Computer engineers are like computer scientists, but are more focused on the hardware side of computers, rather than the theory. Yes, they can both write software, but buy a computer engineer a couple of drinks, and they start talking about AND-gate this, OR-gate that, and how anything can be done with the right hardware.

Meanwhile, over in the corner, a software engineer recommends a more systemic approach to solve the performance problem. "Let's look at what the users need the simulation to do now and where the simulation will most likely grow in the future. Then we can build a computer system that will meet the simulation's computational needs today, and have the hardware architecture to grow in capability as the simulation grows." The computer scientist and the computer engineer turn and look at the software engineer and say: "You sound like an engineer." And there you have it. The strength of a software engineer is in having been trained to apply engineering to the world of computer science. Just as mechanical, electrical, and civil engineers take a methodical approach to solve the problems in their domains, software engineers use the same engineering principles, but in the world of computers.

Bradley T. Hodgins

Team Software Process Coach and Instructor,
Process Resource Team
Naval Air Systems Command

Software Engineering and the Persistent Pursuit of Software Quality

Paul D. Nielsen, Software Engineering Institute

Abstract. Software engineering continually seeks to achieve software quality, as the field keeps pace with the changing role of software. The idea for software engineering was conceived nearly 50 years ago when it was observed that the schedule and performance of software used in IT data systems needed to be improved. Today, as software has become the underpinning of the cyber environment and essential to all aspects of DoD system capabilities and operations, software engineering also encompasses new technologies and practices for cybersecurity that enable informed trust and confidence in using information and communication technology.

Introduction

As it keeps pace with software's strikingly expanding and deepening role for the DoD and the Defense Industrial Base, as well as for civil government and industry more generally, software engineering has remained focused on the cause that launched it—the pursuit of software quality.

At the first NATO Software Engineering Conference in 1968, computer science pioneer Edsger Dijkstra captured a major concern about software-based data systems. “The dissemination of knowledge is of obvious value,” Dijkstra said, “the massive dissemination of error-loaded software is frightening.” The NATO Science Committee invited 50 experts in computer science, such as Dijkstra, to examine a prevailing perception of a “software crisis.” The experts saw problems in the reliability of the large software-based data systems of the day, as well as in their cost and schedule management. Discussion at that conference about the crisis gave birth to software engineering [1]. Software engineering thought leaders subsequently pointed to achieving software quality as a solution for software issues. Watts Humphrey captured the perceptions of many leaders when he defined software quality as “a software product” that “must provide functions of a type and at a time when the user needs them” and “must work” [2].

Response to Software Size and Complexity

While observed in the useful (easy, safe, reliable) operation of a software-reliant system, software quality is determined by practices, tools, technologies, and methods that result from software engineering research and development. Software development methodology, for example, emerged as an area of software engineering research and development beginning in the 1970s, in response to the need to achieve quality in larger software-reliant systems. Incremental or iterative methods, such as the Waterfall approach and its modifications, were explored to determine whether activities that promote software quality are

accomplished throughout the software creation process. In addition, software program managers demanded improved software cost estimation approaches. Research and development by Barry Boehm led to the initial Constructive Cost Model (COCOMO) in 1981. Boehm based his algorithmic estimation model on a study of several dozen projects of varying sizes. He continued to refine his cost estimation tool to keep pace with advances in software development, producing COCOMO II in the 1990s [3].

A decade or so after software development investigation began, the insight that software process improvement could also contribute significantly to software quality became an avenue for research and development. Process improvement places software development into a broader context, highlighting the organization and its practices and processes as a contributing factor to software quality. The idea of a capability maturity model emphasized that the software organization needed to be improved in order to attain software quality. Years of research into the linkages between organizational practices/processes and software quality culminated in 2002 with the establishment of the CMMI® (Capability Maturity Model Integration) framework in work sponsored by the Office of the Secretary of Defense (OSD) and the National Defense Industrial Association (NDIA) and carried out by the software community.

Around the time that the CMMI framework was first released, Agile software development methods also emerged as an alternative development process. Twelve principles underlie Agile software development, including software-quality-centric precepts such as “Working software is the principal measure of progress” and “Continuous attention to technical excellence and good design” [4]. Defense leaders want the agility and velocity that Agile methods bring, but they also often need assuredness and scale that challenge Agile methods. One active area of research now is how to scale Agile to defense-class systems, a goal that may require adjustments in software engineering and acquisition.¹ In addition, Boehm's early work in balancing agility and discipline, his incremental commitment model, and recent work in DevOps are all efforts to find the right blend of these approaches, a blend that might need to be unique to each project and that might vary during the system's acquisition lifecycle [5].

In the 1990s, software engineering thought leaders also began to define concepts and practices in software architecture, a core activity that permits reasoning about properties that enable or inhibit a system's desired qualities, such as availability and security [6]. Dewayne Perry and Alexander Wolf, for instance, examined architecture's place in software process management in 1992 [7]. A few years later, Mary Shaw and David Garlan wrote *Software Architecture: Perspectives on an Emerging Discipline*, introducing key abstractions such as components, connectors, and styles [8]. Subsequent research and development in architecture produced a number of models and methods to describe a software architecture and to evaluate it relative to the system's goals for such quality attributes as modifiability, security, and reliability. Philippe Kruchten, for instance, developed a model to view the architecture called 4+1. Kruchten's model gives a description of the system from the viewpoint of users, project manager, and other stakeholders. His 4+1 views are the logical, process, physical, and development views of the system and the use cases for the system [9]. In addition, the Architecture Tradeoff Analysis Method (ATAM) is a widely used method for architecture evaluation. The

primary output of an ATAM is a set of issues of concern about the architecture. When performed early in the development lifecycle, an ATAM has been shown to help a program avoid costly and schedule-consuming problems that might not emerge until the testing phase or even in fielding [10].

Having grown from an “aspiration” and “rallying cry” [11], software engineering today not only features practices, technologies, tools, and methods for software acquisition and development but also boasts of an established body of knowledge [12] and several international standards. In addition, many colleges and universities offer coursework or degree programs in software engineering, and there are scores of professional conferences annually on aspects of the field.

Software and Cyberspace

However, the march forward in software engineering of practices and standards has not resolved all software quality challenges, because software’s role continues to grow and deepen. The world has become reliant on software-enabled systems and components. In addition, software is now embedded in the cyberspace domain that enables defense military, intelligence, and business operations [13]. As a result, the DoD is keenly aware of the increasing importance of software and the critical need to achieve software quality.

Software is important for the DoD because it promotes lower cost and improved agility in deploying and reconfiguring systems [14]. One result is reflected in the DoD’s ability to now program systems that were once fixed-function to meet changing mission needs. Sensor networks, field programmable gate arrays, software-defined networking, software-defined radios, and embedded controllers represent a few of these now-programmable areas. Another result is that software enables the interconnectivity that is central to accomplishing system-of-systems configurations. Systems of systems support network-centricity, aiding DoD mission goals for information superiority [15]. A third result is that software enables a shift from stovepipe (“platform-centric”) systems to modular (“framework and apps”) approaches [16]. To exploit the flexibility of modular development, the DoD continues to explore the use of an open systems architecture approach that will shift development focus more to payloads and less to platforms.²

The overwhelmingly large role of software in safety-critical air systems (defense and commercial) provides an appropriate illustration. The Air Force vision document *Global Horizons* traces the percentage of capability in air systems reliant on software through generations of aircraft. By the mid-1970s, when the F-16 went into production, software accounted for about 40 percent of capability. A generation later, the F-22 relied on software for 80 percent of capability. Software may contribute 90 percent of capability for today’s premier fighter, the F-35. In addition, millions of lines of software are required to support F-35 Lighting II ground functions [17]. Software’s critical role in delivering capability is driving commercial aircraft makers to seek a new development paradigm. The new paradigm follows an architecture-centric “integrate then build” engineering approach rather than the traditional “build then integrate” one in an effort to reduce software rework costs. In the System Architecture Virtual Integration (SAVI) project, aircraft makers and other organizations (including DoD) created a model of software (development and rework) costs. Based on trends and tradi-

tional development approaches, the SAVI COCOMO II estimate predicted a cost of \$10 billion to develop the millions of SLOC³ required for aircraft built in this decade, an unaffordable amount [18]. SAVI figures also predict that, without a change, software cost will consume an overwhelming portion of total system costs (see Figure 1). In addition, it is not only in development/rework cost that software looms large. Aircraft now remain in use beyond their original expected service lives. A 2011 U.S. Air Force Scientific Advisory Board study found that the cost of software sustainment for defense weapons systems nearly doubled between 2002 and 2011 [19].

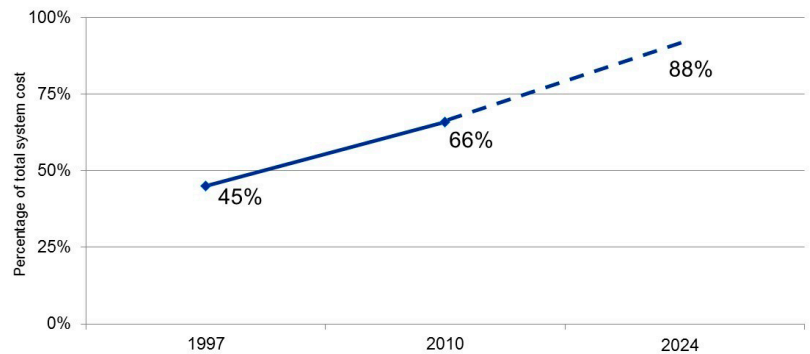


Figure 1. Aircraft Software Development and Rework Cost

With this increased dependence on a software-enabled cyberspace have come new risks and challenges. The size and complexity of software, as well as the interconnectedness of software-enabled systems, mean possible exposure to disruptive, damaging events. Size makes it more likely that software code will include vulnerabilities. Complexity means that organizations may be impacted by emergent behavior—problems almost impossible to foresee during software development or deployment.⁴ Outdated legacy code bases, patches installed too late, new applications added to legacy systems, and interdependencies between systems with different levels of software quality—all could reveal hidden vulnerabilities. Legacy system cybersecurity is an acute concern for the DoD, where critical systems are not easy to modify or patch [20].

When mission-critical systems were standalone entities, security was an afterthought. With software engineering practices, tools, technologies, and methods used to produce complex software that delivers advanced, innovative capabilities that are increasingly integrated and interconnected, cybersecurity can no longer be an afterthought in software engineering.

Cybersecurity Expands Software Quality

Indeed, given the defining role of software in the cyber world, software engineering and cybersecurity are now inseparable. Cybersecurity is now not only one of a software system’s essential qualities but also a factor that expands the meaning of software quality. The pursuit of software quality now also must consider the risks from potential actions of an adversarial/malicious user throughout the software lifecycle (see Figure 2). Cybersecurity needs to be included in activities from the onset of the acquisition, designed and built into the software system, and considered a prime concern as the system is fielded and sustained [21].

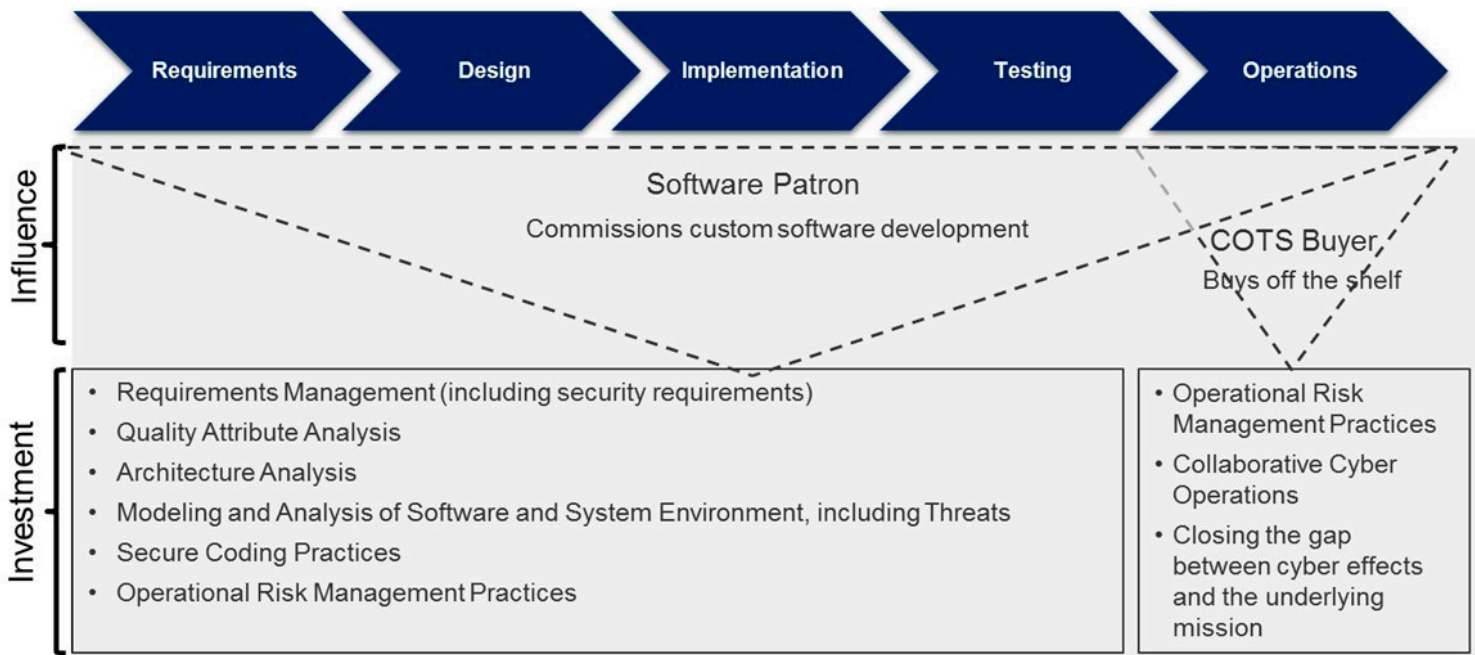


Figure 2. Cyber Risk Must be Addressed Across the Lifecycle for Custom or COTS-based Development

It is vital to approach security requirements in a systematic way early in the lifecycle, in the requirements and design (architecture) phases. Research by Nancy Mead has shown that security requirements can be overlooked or remain implicit until it is much more costly to address them. It is a better practice to perform a risk assessment regarding the system in the context of the expected operating environment and then elicit security requirements [22]. In addition, designing quality architectures involves the use of the fundamental and related concepts—tactics and patterns. Some security tactics or techniques involve detecting, resisting, and reacting to attacks; others aim to help the system recover from attacks [6]. An architectural security pattern is a “piece of design” that provides a proven solution for achieving a particular quality attribute. A typical pattern for system security is authentication and authorization [6]. Including cybersecurity concerns early in development will pay off later on in terms of software quality that is reflected in reliability and maintainability, as well as in user satisfaction.

Likewise, it is important to evaluate for cybersecurity during coding and testing activities. Many exploitable software vulnerabilities occur because of common coding errors. MITRE Corporation sponsors and maintains the Common Weakness Enumeration (CWE) dictionary, under the leadership of Robert Martin. The “common” software flaws, faults, and other errors in code, design, architecture, or implementation in the CWE could result in vulnerabilities others will exploit [23]. These weaknesses—such as buffer overflows, authentication errors, and insufficient data validation—are likely to be as easy to find and mitigate as they are to exploit. Eliminating common vulnerabilities during software development can result not only in more secure software but also in a large cost reduction, because less effort will be expended to repair code. Government, industry, and academic cybersecurity researchers are forming and promoting the adoption of international secure coding standards for some common software programming languages, including C, C++,

Perl, and Java⁵ [24]. It is important to prevent errors through adherence to secure coding standards; however, rigorous testing is also advisable. For instance, vulnerabilities may emerge as software components are integrated, in commercial off-the-shelf (COTS) and custom-developed software, or in patches sent out to eliminate already discovered vulnerabilities. An advanced level of software testing would include full penetration testing by organic or external experts.

Cybersecurity concerns for software quality must also account for a software supply chain that is diverse and complex—even global. Consider the variety in these supply chains: physical components, integrated components such as network routers, software, the prime contractor organization, subcontractor organizations, and other supply chains for the commercial products used [25]. Each component might be deemed to have sufficient quality, but the integration of components with different levels of software quality ratchets up cybersecurity—and mission—risk for the system [13]. The complicated software supply chain has become an avenue for cyber intrusions, as well. For example, in 2014 a counterfeit, malware-containing Netflix app was pre-installed somewhere in the supply chain on new Android devices available from several vendors [26].

The introduction of malware by a supply chain partner also suggests insider threat concerns. Recent high profile incidents such as Edward Snowden’s actions and the Target Corporation breach heighten awareness of the threat that insiders (malicious or unintentional) pose from fraud, sabotage, or theft of intellectual property. While Snowden, working as an NSA contractor, appears to have acted intentionally, the theft of credit card information from Target is reported to have resulted from a mistake by an employee at a supplier that had access for electronic billing to the firm’s network [27]. For more than a decade, the CERT Insider Threat Center—collaborating with the DoD, U.S. Department of Homeland Security, the U.S. Secret Service, other federal agencies, the intelligence community, private industry,

academia, and the vendor community—has researched insider threats and built tools for their mitigation. An understanding of insider threat mitigation is especially important for U.S. government agencies working to establish programs to meet Executive Order 13587 requirements.

Finally, despite efforts to assure software quality by preventing software vulnerabilities in development or patching them in system operation, and even with stepped-up insider threat monitoring, it is prudent to assume that systems may be under attack. In operation, a software system may be vulnerable to attack through the exploitation of previously unknown software vulnerabilities (zero-day attack), intrusion into a communications channel (man-in-the-middle), infection of a website visited by a targeted user (watering hole), and other avenues. Thus, an overarching aspect for software quality in the cyberspace domain is operational system resilience. It is appropriate to ask, “What truly needs to be protected? Even when compromised, can the software system continue to deliver capabilities that users need, when they need them?”

Software Quality is a Constant Purpose and Software is a Moving Target

The goal to provide software that must “provide functions of a type and at a time when the user needs them” and “must work”

is a fixed point in the software universe. However, the changing and expanding role that software plays in cyberspace means that software engineering has to continue to evolve (even leap ahead) in the ongoing pursuit of software quality. Software engineering now needs to be proactive, not reactive.

When Dijkstra set off an alarm about “the massive dissemination of error-loaded software,” systems relying on software touched DoD and other organizations in much more definable ways. Today, military, civil government, industry, and society more generally communicate and socialize in an environment that relies on software-reliant global IT architectures, applications, and services. It is inevitable that demands from users, program managers, and developers for software that delivers greater functionality, reliability, performance, security, autonomy, maintainability, and a host of other attributes will spur innovation and new paradigms that today are not yet conceived. In addition, research and technology trends (see some examples in Table 1) will continue to build on current levels of software complexity and capability. The Internet of Things (IoT), for example, is rapidly approaching, if not in place in some sectors. Already one can see steady progress toward realizing IoT constructs such as the smart grid, smart cities, and smart homes. People and technology will push the frontiers of software engineering forward.

Architecture	Cybersecurity	Process	Workforce	Market
Complexity	Global supply chain security	International standards	Globalization of software development capability	Internet of everything
Cyber-physical systems	Secure coding practices & tools	Data-driven decision-making about practices to use	Supply and demand issues	Autonomy
Strategies for technical debt	Automated software vulnerability discovery	Continuous delivery/velocity	Talent management	Big data/analytics
Affordable sustainment/evolution	Network situational awareness	Blending development and operations	Skills for managers and boards	Software-defined environments
Socio-adaptive systems	Insider threat mitigation	Improving early lifecycle cost estimation	Continuous education	Consumerization
Modeling/virtual integration	Malware analysis & databases	Model-based engineering and auto-code generation tools		Development velocity
Interoperability	Cyber intel for risk management	Assurance planning		
	Adaptive intrusion detection and remediation			
	Active defense			

Table 1. Selected Software Engineering Research Trends

Pursuing software quality in the highly connected (bordering on hyper connected) cyber world may present software engineering with different new frontiers, as well. It could call for greater appreciation of software quality outside the realm of software professionals (developers, architects, programmers, and the like). How can software engineering encourage software quality through broader education at all levels of an organization? For example, what do all employees in an organization now need to know about building a business case for new or updated software systems, securing the global software supply chain, information security/insider threat, appreciating what determines system behavior, or the use of Agile development approaches? Or, how should senior executives incorporate software in their risk assessments?

As software's size, complexity, security and interconnectedness grow, the role of software engineering will become more fundamental to the entire system lifecycle and system-of-systems integration. Software engineering has advanced significantly in its first 50 years, but the continued search for more integrated capabilities opens new opportunities and challenges for researchers, practitioners, and users. ❖

Disclaimers:

Copyright 2014 Carnegie Mellon University

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

This material has been approved for public release and unlimited distribution.

ATAM®, Capability Maturity Model®, Carnegie Mellon®, CERT®, CMM® and CMMI® are registered marks of Carnegie Mellon University. DM-0002001

ABOUT THE AUTHOR



Paul D. Nielsen is Director and Chief Executive Officer of the Carnegie Mellon University Software Engineering Institute (SEI). Prior to joining the SEI, Nielsen served in the U.S. Air Force, retiring as a major general and commander of Air Force research. Nielsen is a member of the U.S. National Academy of Engineering and a Fellow of both the American Institute of Aeronautics and Astronautics and the Institute for Electrical and Electronics Engineers. He has served on many government boards and advisory groups and is currently a member of the Defense Science Board.

Phone: 412-268-5800

E-mail: nielsen@sei.cmu.edu

NOTES

1. See Table 1 for a list of some current trends in software engineering research areas.
2. See, for example, Douglas Schmidt's blog about the Navy's Open Systems Architecture Initiative <<http://blog.sei.cmu.edu/post.cfm/importance-automated-testing-open-systems-architecture-062>>.
3. Source lines of code
4. See, for example, Douglas Schmidt's blog about the Navy's Open Systems Architecture Initiative <<http://blog.sei.cmu.edu/post.cfm/importance-automated-testing-open-systems-architecture-062>>.
5. Java is a trademark of Oracle, Inc.

REFERENCES

1. Peter Naur and Brian Randell, eds. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, Brussels, Scientific Affairs Division, NATO, January 1969, pp. 17, 70. <http://homepages.cs.ncl.ac.uk/~brian.randell/NATO/nato1968.PDF>
2. Watts S. Humphrey. *A Discipline for Software Engineering*. New York: Addison-Wesley, 1995, p. 272.
3. Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software Cost Estimation with COCOMO II*. Englewood Cliffs, NJ: Prentice-Hall, 2000.
4. Mary Ann Lapham, Suzanne Miller, Lorraine Adams, Nanette Brown, Bart Hackemack, Charles (Bud) Hammons (PhD), Linda Levine (PhD), and Alfred Schenker. *Agile Methods: Selected DoD Management and Acquisition Concerns (CMU/SEI-2011-TN-002)*. Pittsburgh, PA: Carnegie Mellon University: Software Engineering Institute, 2011, pp. 1, 16 <http://www.sei.cmu.edu/reports/11tn002.pdf>
5. Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Professional, 2003
6. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice: Third Edition*. New York: Pearson Education, 2013, p. 26.
7. Dewayne Perry and Alexander Wolf. "Foundations for the study of software architecture," *ACM Sigsoft Software Engineering Notes*, vol. 17, no. 4, pp. 40-52, 1992.
8. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ (USA): Prentice Hall, 1996.
9. Philippe Kruchten. "Architectural Blueprints—the "4+1" View Model of Software Architecture." *IEEE Software*, vol. 12, no. 6, pp. 42-50, 1995.
10. Robert L. Nord, John Bergey, Stephen Blanchette, Jr., and Mark Klein. *Impact of Army Architecture Evaluations (CMU/SEI-2009-SR-007)*. Pittsburgh, PA: Carnegie Mellon University Software Engineering Institute, 2009.
11. Mary Shaw. *Prospects for an Engineering Discipline of Software (CMU-CS-90-165)*. Pittsburgh, PA: Carnegie Mellon University, September 1990.
12. IEEE. *SWEBOK v3.0*. <http://www.computer.org/portal/web/swebok>
13. Department of Defense (DoD). *Department of Defense Strategy for Operating in Cyber space*. Department of Defense, 2011. <http://www.defense.gov/news/d20110714cyber.pdf>
14. National Research Council. *Critical Code: Software Producibility for Defense*. Washington, DC: National Academies Press, 2010.
15. David S. Alberts, John J. Gartska, and Frederick P. Stein. *Network Centric Warfare: Developing and Leveraging Information Superiority: Second Edition*. Washington, DC: CCRP, 2000. http://www.dau.mil/pubscats/ATL%20Docs/Mar_Apr_2013/Hagen_Sorenson.pdf
16. ADM Jonathan W Greenert, U.S. Navy. "Payloads over Platforms: Charting a New Course," *Proceedings Magazine*, 138/7/1313, 2012. <http://www.usni.org/magazines/proceedings/2012-07/payloads-over-platforms-charting-new-course>
17. U.S. Air Force. *Global Horizons: United States Air Force Global Science and Technology Vision (AF/ST TR 13-01)*. U.S. Air Force, 2013; also Christian Hagen and Jeff Sorensen. *Delivering Military Systems Affordably, Innovations in Systems and Software Engineering*, vol. 7 issue 3, 2011, p. 161-170 http://www.dau.mil/pubscats/ATL%20Docs/Mar_Apr_2013/Hagen_Sorenson.pdf
18. D. Ward and S. Helton. "Estimating Return on Investment for SAVI (a Model-Based Virtual Integration Process)," *SAE International Journal of Aerospace*, vol. 4, no. 2, pp. 934-943, 2011; Peter Feiler, Jörgen Hansson, Dionisio de Niz, and Lutz Wrage. *System Architecture Virtual Integration: An Industrial Case Study*, Pittsburgh, PA: Carnegie Mellon University Software engineering Institute, Technical Report CMU/SEI-2009-TR-017, 2009. <http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9145>; and Peter Feiler. *An Incremental Life-Cycle Assurance Strategy for Critical System Certification*, Proceedings and Presentations of the TSP Symposium, November 2014. <http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=423668>
19. U.S. Air Force Scientific Advisory Board. *Sustaining Air Force Aging Aircraft into the 21st Century (SAB-TR-11-01)*. U.S. Air Force, 2011. <http://www.dtic.mil/cgi-bin/GetTRDoc?Location=U2&doc=GetTRDoc.pdf&AD=ADA562696>
20. Joint Publication 3-12 (R). *Cyberspace Operations*. 5 February 2013. http://fas.org/irp/doddir/dod/jp3_12r.pdf
21. Defense Science Board. *Mission Impact of Foreign Influence on DoD Software*. Washington, DC: Office of the Under Secretary of Defense for Acquisition, Technology, and Logistics, September 2007. <http://www.acq.osd.mil/dsb/reports/ADA486949.pdf>
22. Nancy Mead. *SQUARE Process*. Pittsburgh, PA: Carnegie Mellon University Software engineering Institute, 2006.
23. MITRE Corporation. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. <http://cwe.mitre.org/top25/>
24. CERT. *Secure Coding*. <http://www.cert.org/secure-coding/>
25. Robert J. Ellison, Christopher Alberts, Rita Creel, Audrey Dorofee, and Carol Woody. *Software Supply Chain Risk Management: From Products to Systems of Systems (CMU/SEI-2020-TN-026)*. Pittsburgh, PA: Carnegie Mellon University Software Engineering Institute, 2010.
26. Jeremy Kirk. "Pre-installed malware turns up on new phones." *PC World*, March 4, 2014. <http://www.pcworld.com/article/2104760/preinstalled-malware-turns-up-on-new-phones.html>
27. The Editorial Board of the NY Times. Edward Snowden, Whistle-Blower. January 1, 2014. http://www.nytimes.com/2014/01/02/opinion/edward-snowden-whistle-blower.html?_r=0 and Marcelo Ballea and John Heggsteuen. *Target's Data Breach Began with a Company that Does Heating and A/C Work on Its Stores*. *Business Insider*. February 7, 2014. <http://www.businessinsider.com/target-data-breach-explained-2014-2>



Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C seeks dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer & Electronic Engineering
- Digital Forensics
- Telecommunications Assurance
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to www.dhs.gov/cybercareers. To apply for a vacant position please go to www.usajobs.gov or visit us at www.DHS.gov.

Enough Software Processes, Let's Do Patterns!

Paul E. McMahon, PEM Systems

Abstract. This paper explains common mistakes many organizations have made in the past when implementing defined processes with a goal of improving software practitioner performance. The paper suggests a more effective approach to improvement is to focus on patterns where software practitioners need help in making better decisions. A practical approach to develop patterns is described along with multiple pattern examples, including examples that can help software practitioner supporting roles.

Introduction

What is software development? Is it a craft, or an engineering discipline? Regardless of how you view it, there are over 11 million practitioners developing software today [1]. Therefore a more practical question is:

How can we do better at helping software practitioners with the common challenges they face each day?

One approach that has helped is reusing design patterns [2]. However, software development extends far beyond just design and programming. In fact one organization reported that their typical software developer's workday breaks down as follows [3]:

- Design— 0.5 hour
- Programming— 1.0 hour
- Requirements gathering, analysis— 0.5 hour
- System testing, fixing bugs— 2.0 hours
- Deployment support— 0.5 hour
- Project management, walk-throughs— 0.5 hour
- Admin tasks (e.g. reading email, meetings etc.)— 3.0 hours

This means that for every 1.5 hours of design and programming, there is another 3.5 hours of work for other common software development activities. These numbers are not unusual, and some might argue even with this breakdown many organizations are not spending enough time on requirements, and testing.

This raises another question: Are there other patterns beyond design and programming that could help software practitioners with the other challenges they face each day?

The idea of patterns beyond design and programming is not completely new. Scott Ambler has written extensively about the potential value of process patterns [4]. However, to date, patterns have not received anywhere near the attention they deserve given their potential value to improve both personal and organizational performance.

Motivation for Patterns Beyond Design and Programming

The exact amount of time a software developer spends beyond design and programming varies across different organizations. However, even in organizations where requirements gathering is handled by business analysts and customer interactions are handled by marketing, software developers still spend significant time clarifying ambiguous requirements and interacting with stakeholders.

Furthermore, with today's emphasis on self-directed teams, estimating the work, reporting status, and identifying and working improvements to tools and practices all fall within today's typical software developer's task list. This leads to another question: Is it possible that helping software professionals with pattern recognition beyond design and programming holds great untapped potential for high value personal and organizational performance improvement?

When you consider how much time software practitioners are spending beyond design and programming this potential is difficult to ignore.

What Do We Mean By a Pattern?

To understand the value of patterns let's start with an analogy. We've known for a long time that the best way to build a complex software system is to break the problem down into small chunks and build and deploy those small chunks incrementally in small slices. This approach works because it gets product into the hands of customers sooner where those same customers can provide rapid feedback ensuring the software development team stays on the right track.

So if it makes sense to build and deploy our software systems in small slices to keep developers on track, why wouldn't it make sense to build and deploy the practices we want our developers to follow in small slices to ensure we are giving them the help they need to stay on track?

When I refer to a "practice slice" I mean a common scenario or a small set of related scenarios that a software practitioner typically faces each day.

When I refer to a "pattern" I mean the essentials of a practice slice with less important details left out, and key related questions, tips, and warnings added in to help practitioners make better decisions when facing similar situations [5].

More Motivation for Patterns

What do many organizations do when a new software practitioner comes into the company? They want the new practitioner to learn the "way-we-do-software". So they give them the volumes of their "process documentation," and they tell them to study it because it contains "how-we-expect-you-to-operate". While some of this is necessary, there is another side to consider.

What if you viewed your software practitioners as your customer for your organizational processes?

By looking at it this way, rather than thinking of your company processes as just direction to your practitioners, they also become a vehicle to help you listen and respond better to your

practitioner's needs. When organizations focus on patterns they listen better to the common situations that their practitioners are struggling with. This can generate valuable discussions with experienced practitioners helping less experienced practitioners learn faster how to handle these difficult situations. This can lead to high value and rapid performance improvement.

I think most would agree that our processes should help practitioners perform better. However, where people tend to disagree is on the best path to get to this goal. Now let's discuss more about past mistakes.

Process Deployment in the Past

Practitioners face all kinds of different situations each day and so our processes can't possibly tell practitioners what to do in each situation.

One mistake many organizations have made in the past is trying to define in detail what practitioners should do in every situation. This has led in some organizations to processes that are so heavyweight that they are not usable by human beings. Other organizations have gone to the other extreme making their processes so light as to become trivial and of minimal, or no, value. What practitioners really need falls between these two extremes.

Where Software Practitioners Need Help the Most

There are two types of information most software practitioners need. First is basic information about a process, such as how to conduct it, competency needs, checklists to prepare, and completion criteria. This kind of information is useful primarily for beginners. Traditional process definitions have done a reasonably good job at meeting this need.

The second type is the information that practitioners need while they are executing the process. This includes things to watch out for such as common situations where they may need to make a decision. This may include trade-offs between options they have. This is where most practitioners—once they move beyond the beginner's stage—need help the most, and this is where patterns can help the most.

Implementing the Most Valuable Patterns to Aid Performance

When developing patterns to aid decision-making what is most important is to NOT pick just any common scenario. For many common situations that happen everyday practitioners don't need extra help. Therefore you want to pick the situations where your practitioners often fail to make good decisions. This increases the value of the pattern as it helps the practitioner where they need help the most during a typical day.

I have developed patterns with a number of clients. First we examined the common scenarios their practitioners were facing each day. Then we prioritized the scenarios and selected a small set where there were tendencies toward poor decisions that hurt performance. [5]

We then added in key questions practitioners should be asking, tips and warnings based on further discussions with experienced practitioners. Then we trained the people in the patterns, and we gave them simple aids they could take back to use as on-the-job reminders.

Patterns are a simple and practical way to communicate to less experienced practitioners how the more experienced practitioners in your organization would handle the common difficult situations they are likely to face.

Pattern Examples

Following are three simple pattern examples, specifically for software developers, created by expressing scenario essentials in a user story form [6], and adding in related questions practitioners should be asking, tips based on the answers to the questions, and warnings.

Pattern One: "I don't Understand a Requirement"

User Story: "As a software developer, I want more guidance in what I should do when I don't understand a requirement, so I can build the best software in the least amount of time to meet my customer's needs and my commitments."

Questions practitioners should be asking:

- Is my customer collaborative, and working closely with me to discover the requirements?
- Do I have a fixed schedule and cost?

Tips:

- Do nothing different if your customer understands the cost of iterating on requirements and is working closely with you to discover the requirements.
- Raise a risk if your customer is non-collaborative and you have a fixed cost and schedule.

Pattern Two: "My Testing and Peer Reviews are Taking Too Long"

User Story: "As a software developer, I want to know what to do when my testing and peer reviews are taking too long, and I am getting pressure from my manager to finish on time."

Questions practitioners should be asking:

- Does my software have life-critical consequences if it fails?
- Does my project have an agreed way of working with respect to testing? If so, have I reviewed my testing process to see what options I have?
- Does my peer review process provide options for focused, streamlined peer reviews based on criteria? If so, have I considered those options?

Tips:

- Consider focusing your low level testing just on areas you have changed, if allowed by your agreed way of working.
- Consider focusing tests and peer reviews on high risk areas, if allowed by your agreed way of working.

Warning:

- Be sure to assess any risk involved in reduced testing or streamlining peer reviews, such as:
- Missing key dependencies.
- Not following the agreed way of working.

Pattern Three: “How Should I Handle a Design Risk?”

User Story: “As a software developer, I want more help in how to handle a design risk, when the alternative designs are going to extend the schedule and I am getting pressure to finish on time.”

Questions practitioners should be asking:

- Have I discussed my design with a more experienced coworker or colleague who has implemented a similar design?

Tips:

- Call for a peer review.
- Call for a limited peer review with key people focusing just on the design risk, if allowed by your agreed way of working.

Some of these questions and tips may seem obvious, but keep in mind that practitioners tend to forget the obvious under project pressures. One of the key values that patterns provide over simpler checklists is they give your practitioners context and remind them of choices they have, but might have forgotten.

It is critical for practitioners to think about their own context when making on-the-job decisions and patterns can aid practitioner critical-thinking in ways that traditional checklists and process definitions sometimes fall short. When developing your own patterns be sure to conduct discussions with your experienced practitioners so the questions, tips and warnings are relevant to your own organization's context, constraints, and culture.

More Sample Scenarios and Guidance To Kick-Start Your Own Pattern Development

Below are three more sample scenarios with related questions, tips, warnings, and some additional guidance that can help kick-start your own pattern development. The scenarios provided are all based on real experiences I have observed with actual clients and I have found that these types of scenarios tend to repeat in many organizations¹ [5].

The patterns in this section are not limited to software developers. I have included patterns that may be of interest to software practitioner supporting roles such as coaches, software testers, team leaders, Scrum Masters, project managers, and process improvement professionals. I have included these additional patterns to demonstrate how patterns can be used to aid the performance of software supporting roles as well software developers.

The scenarios in this section include additional context information beyond the simpler user story form provided previously. This additional information may be beneficial for practitioners when first learning the scenario in training, whereas the simpler user story form may be sufficient as a quick on-the-job reminder. The related questions have been developed based on my own consulting experiences, and the use of the Essence framework [7, 8, 9, 10, 11].

Pattern Four: “Getting Your Coaches on the Same Page”

Coaches can help a team, but multiple coaches helping the same team need to coach consistently. This is particularly important when teams are transitioning to a new agreed way of working as conflicting advice can do more harm than good. This scenario should be

of interest to coaches, Scrum Masters, team leaders, developers, testers and process improvement professionals.

Scenario start:

I was coaching a team that was moving from a traditional waterfall development approach to an agile-Scrum approach. This company had a history of being run by PMI certified project managers requiring detailed work breakdown structures up front with detailed tasks assigned and tracked by all project managers.

When I first explained to the practitioners in the training class the concept of self-directed teams, everyone liked the approach and seemed to understand it. But after the training the team had trouble following the new process. After the first few daily stand-up meetings [12] the team members would go back to their workstations and sit waiting for someone to tell them what to do next. This was because the culture in this organization was for people to just do what they were told.

Another reason the practitioners had trouble following the new practices was because some of the new practices seemed to conflict with the existing organizational policies, as well as the existing culture. For example, testers in this organization were not allowed to develop code.

The project manager, who was learning to be a Scrum Master, expressed concern about the situation and wanted to know if she should re-institute her two hour weekly meeting where she had traditionally assigned tasks to individuals and driven actions to closure. I replied:

“No, this is not your role now. But you do need to remind the team members of the new expectations whenever you observe a problem, and you and I need to talk more to make sure we are coaching consistently and in a way that fits within your company constraints.”

This was a style change for the project manager, and it required some time for the project manager and myself to work through a number of issues related to constraints on the team's operation.

This team had competent team members committed to start the mission, but they weren't yet collaborating given the new agile-Scrum practices.

Once we worked through these issues and communicated practice clarifications to the team, it didn't take long before the team was collaborating in a self-directed way.

Questions Teams Leaders and Process Improvement Professionals Should Be Asking:

- If you are using multiple coaches, such as an external coach and an internal coach, have you worked through any issues that might exist with your agreed to way of working (e.g. organizational policy constraints) to ensure consistent coaching?

Tip to Team Leaders:

- Consider using a head-coach at least at the start of the endeavor to coach the coaches ensuring key points— especially those associated with likely transition trouble spots—are understood and coached consistently.

Examples of typical areas where issues might arise when transitioning to an agile-Scrum approach include [13]:

- Constraints on who can sign up for certain task types
- Capacity planning expectations
- Risk assessment expectations
- Expectations on basis of task estimates
- Expectations on use of team velocity
- Expectations on use of burn-down charts

It is not uncommon in organizations moving from a traditional command and control approach to a self-directed team approach to need reminders and clarifications on basic expectations during the initial transition. This is a culture change in many organizations and the degree of agility that makes sense in one organization often differs substantially from another due to project specific conditions. Therefore it is critical for each team and their coaches to agree on their way of working and then coach consistently to those agreements.

Warning:

Keep an eye out for coaches who just coach to their own experiences and aren't attune to the needs of your organization.

Pattern Five: The “Non-Engaged Stakeholder”

In Pattern One we saw an example where a stakeholder does not know their own requirements. In this scenario we examine a different type of problem with a stakeholder. This is the situation when your stakeholder knows the requirements, but isn't providing the information you need in a timely fashion. This scenario should be of interest to all software practitioners, project managers, and subcontract managers.

Scenario start:

On the LEM project Don's company was subcontracting component testing. Critical test data was required to be supplied by a customer agency, but the agency kept putting off Don's repeated requests for the data. Don was the project manager and this situation put his subcontractor over budget and behind schedule. Eventually the data arrived and Don approved additional budget for his subcontractor to complete the testing. However, since this project was at a fixed cost to the prime contractor, it resulted in significant lost profit for Don's company.

Tip:

When you face this kind of dilemma consider first digging to see if you can uncover the root cause.

Questions the project manager and subcontract manager should be asking:

- Has a stakeholder representative for the customer agency been appointed and has he agreed to take on the responsibility to provide the required data?
- If so, what is causing the lack of responsiveness?

Tip:

- Often in these cases a stakeholder representative has not been appointed within the responsible organization, or the representative doesn't realize the impact caused by their lack of responsiveness.

There are many other possible reasons why dependent stakeholders fail to respond in a timely way. Examples include:

- Too much work on their plates
- Lack of communication
- Lack of understanding of the exact need
- Poor processes within their own organization
- Lack of authorization

With today's move to more agile approaches there is increased demand for stakeholder representative resources. Often dependent organizations, unless they are already working in an agile way, do not have adequate resources to respond in a timely way. Your first goal should always be to figure out the root of the problem so proper corrective action can be taken.

Warning:

Keep on the lookout for stakeholder representatives who may lack authorization to perform their responsibilities.

Pattern Six: “Are We Really Getting Better?”

This pattern demonstrates how a team can easily fall into the “going through the motions” trap by not making value-added improvements that can aid their performance. This scenario should be of interest to all software practitioners, coaches, Scrum Masters, and process improvement professionals.

Scenario start:

I was asked to help a company that was having trouble hitting their schedule commitments. The company was using an agile-Scrum approach on most of their projects. I started my investigation by sitting in on a retrospective [12] listening to the team discussion.

The improvements they were planning to put in place during the next sprint surprised me because I didn't hear anything about schedule problems being addressed. So I challenged the team by asking the following question: “Why aren't you hitting your schedule commitments?”

One developer immediately responded: “Because our product backlog is never properly prepared. We always end up doing more work than planned because the requirements are always vague and incomplete.”

Question Scrum Masters and Software Practitioners Should Be Asking:

- If your team is using a Scrum approach, are they seriously addressing the real problems on their project in their retrospective discussions?

Scenario continued:

I challenged the team by suggesting that they should reject backlog entries that are not properly prepared. However, a number of team members objected to my suggestion saying that it would never work in their organization. They said that management expected them to do whatever it takes to get the job done and rejecting requirements is not an option.

I then suggested that they should start keeping personal data on the time they spent addressing poorly prepared backlog items, as well as the time they spent designing, coding and test-

ing. I also suggested that when they estimate tasks in the future they should add in the backlog preparation time, and this would help them hit their estimates more accurately in the future.

I further suggested that they use the personal data they collected as rationale in case anyone objected to their estimate, and that they should not reduce their estimate until they had more current data that indicated their schedule performance had improved.

Tip/Warning to Coaches, Scrum Masters and Team Leaders:

- Scrum teams can easily fall into a “going through the motions” syndrome especially during their retrospectives. Scrum teams must continually tackle the hard problems they have seeking out high value performance improvements. Coaches should constantly be on the look-out for the “going through the motions” sign and continually challenge the team to seek higher performance.

Conclusion

Patterns don't replace your company processes, or what your team is doing today. They are an aid that can help your team's critical-thinking and decision-making. They can help team's make better decisions by reminding them of common scenarios, questions to ask, conditions to be aware of and options they have that can ultimately lead your organization to higher sustainable performance.

I encourage you to develop your own patterns to help guide your software developers in making better decisions. I also encourage you to involve your own experienced practitioners in developing your own scenarios, relevant questions, tips and warnings based on your specific context, organizational constraints, and culture. If you are unsure whether patterns can help your practitioners beyond whatever you are doing today, consider polling your practitioners to get the opinion of those you seek to help. ✎

ABOUT THE AUTHOR



Paul E. McMahon, Principal, PEM Systems (www.pemsystems.com) has been an independent consultant since 1997. He has published more than 45 articles and multiple books including “15 Fundamentals for Higher Performance in Software Development.” Paul is a Certified Scrum Master and a Certified Lean Six Sigma Black Belt. His insights reflect 24 years of industry experience, and 17 years of consulting/coaching experience. Paul has been a leader in the SEMAT initiative since 2010.

Phone: 607-798-7740

E-mail: pemcmahon1@gmail.com

NOTES

1. For more information on patterns and pattern examples refer to the book, “15 Fundamentals for Higher Performance in Software Development.”

REFERENCES

1. IDC Study: How Many Software Developers Are Out There? <<http://www.infoq.com/news/2014/01/IDC-software-developers>>
2. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
3. For every hour cutting code, how many hours are spent doing something else? <<http://pm.stackexchange.com/questions/10116/for-every-hour-cutting-code-how-many-project-hours-are-spent-doing-something-e>>
4. Ambler, Scott, An Introduction to Process Patterns, <<http://www.ambysoft.com/downloads/processPatterns.pdf>>
5. McMahon, Paul E, 15 Fundamentals for Higher Performance in Software Development, PEM Systems, 2014,
6. Cohn, Mike, User Stories Applied, Addison-Wesley, 2004
7. Jacobson, Ivar, NG, Pan-Wei, McMahon, Paul E, Spence, Ian, Lidman, Svante, The Essence of Software Engineering: Applying the SEMAT Kernel, Addison-Wesley, 2013
8. Jacobson, Ivar, Ng Pan-Wei, McMahon, Paul, Spence, Ian, Lidman, Svante, The Essence of Software Engineering: The SEMAT kernel, Oct, 2012, ACMQueue, <<http://queue.acm.org/detail.cfm?id=2389616>>
9. OMG Essence Specification, <<http://www.omg.org/spec/Essence/Current>>
10. SEMAT web site, <www.semat.org>
11. McMahon, Paul E, Integrating CMMI and Agile Development: Case Studies and Proven Techniques for Faster Performance Improvement, Addison-Wesley, 2011
12. Sutherland, Jeff, Schwaber, Ken, “Scrum Guide– The Definitive Guide to Scrum: The Rules of the Game”, July 2013
13. Cohn, Mike, Agile Estimating and Planning, Pearson Education, 2006.

The Impact of Contextual Factors on the Security of Code

Dan Shoemaker, Ph.D., University of Detroit Mercy
Carol Woody, Ph.D., Software Engineering Institute

Abstract. Non-technical decisions made in policy, acquisition, governance, resources, processes, and every other aspect of managing software have a direct impact on the resulting operational security. However, these relationships are hidden because the structures we use to govern and organize software do not highlight the security decisions made throughout the life cycle and connect them to the ultimate results. As a result of this obscurity, seemingly appropriate choices result in unacceptable operational security risks because none of the participants recognize the cause and effect linkages.

The Importance of Contextual Factors

Software security is a critical topic. That is because a single key flaw in a major component can bring down the entire infrastructure. But when we think of software and its security we generally think about it in terms of the software engineering technology, people and processes that are involved in its production and sustainment. We rarely think of the development and maintenance of a system or software artifact within the larger context of how it was acquired, resourced, evolved, or managed. Nonetheless, those larger issues have significant real impacts on the security of every system.

It is reasonable to view software security issues strictly through the lens of the software engineering process. The discrete activities of the development and maintenance phases are well-understood and represent the orthodox understanding of the software industry. They are also the places where flaws are actually created. The activities of global product acquisition, strategic planning, resourcing, and overall business process alignment all take place in the realm of the business outside of the traditional lifecycle. As a result, the relationship between those activities and defects in code are less clear.

Nonetheless, the specific context in which a system is resourced, overseen, managed and assured will have a lot to do with how successfully it performs in actual practice. Software is only as good as the people, processes and tools that produce it. The criticality of the development context as introduced by Watts Humphrey in his 1989 publication [1] and further described as a key aspect of the software engineering discipline in his later publication [2] should be considered a fundamental tenet of the software engineering profession. That necessary coordination of context requires a perspective that is more in the realm of business strategy than it is technological. And in that regard, if there are disconnects between

the technical processes and the relevant elements of the business operation there is a potential for the injection of serious exploitable vulnerabilities in the actual product.

It is well documented that coherent, well-defined and effective strategic processes are a factor in the production of software and systems. The Software Engineering Institute (SEI) created model after model in the 1990s to underwrite capability [3,4,5] as a critical element of effective software production. Likewise, SEI currently utilizes three different large-scale approaches; two which characterize the capability maturity of the overall process [6,7]; as well as another to describe the strategic maturity of services [8]. That is not to mention the various models of process capability that have been produced by the people at ISO [9,10].

This body of evidence leads us back to the initial premise, which is that the larger context is going to directly impact the correctness, and by implication the security, of every system, or software product. Greater attention has to be paid to contextual factors, which have been largely ignored in the debate about "why Johnny can't code."

Security Impacts of Contextual Factors

Attention has to be given to the impact of business strategy, organizational controls, business process alignment and strategic resourcing decisions on the resulting software and its security.

Business Strategy

The elements of business strategy that impact software and system security include such items as acquisition outsourcing decisions. But they also comprise decisions about organizational development strategy, staffing and training policy.

From a security perspective, the most important choice that an organization is likely to make is the buy-versus-make decision. Many business advantages are associated with commercial-off-the-shelf (COTS) products [11]. Nevertheless, the General Accounting Office lists five areas of high risk in a COTS strategy [12]. Those are: 1) malware, 2) counterfeits, 3) supply chain breakdowns, 4) supplier incapability, and 5) software defects. All of those areas can introduce major security concerns into the organization's electronic infrastructure. Consequently, an intelligent supply chain risk management strategy is an essential component of good system and software security practice.

In addition to supply chains there are the risks associated with the organization's acquisition process itself including specification failures, changing requirements, time and cost over-runs due to lack of control of the process, and insecure product selection [13]. All of these potential issues have to be factored into the outcome of the acquisition process. And very few of these difference-making factors are seen as being a consideration of the acquisition strategy itself. The acquisition strategy chosen to divide system components among several vendors, mandate a reliance on small businesses, rely on commercial off the shelf products (COTS), include open source (or not), purchase from foreign vendors, rely on legacy, and leverage existing hardware infrastructure just to mention a few of the options each contribute to the resulting attack surface, interfaces, and operational security capabilities of the resulting implementation.

These all have real consequences with profound security implications. And thus they are all valid areas of long-term security concern. An organization that allows promiscuous, or unmanaged outsourcing activity is literally playing Russian roulette with the likelihood of system, or software exploitation. Accordingly the points of failure associated with system and software acquisition all have to be thought about and dealt with as part of the overall process of ensuring the integrity and trustworthiness of an organization's electronic infrastructure [14].

The actual technical work also requires a larger perspective. Organizational culture is shaped by strategic policies, which are explicitly defined by the organization in order to ensure the proper level of employee motivation, discipline and training. The Software Engineering Institute issued a model focused entirely on the importance of people to the software development effort [4].

The requisite motivation, discipline and level of skill have to be explicitly fitted to the overall organizational mission and maintained as such. That includes documenting the performance expectations for the work to be done in order to ensure clarity, the promulgation of those expectations to all members of the organization in order to ensure common understanding, and the monitoring of employee performance in order to ensure effectiveness. Anybody who thinks that the culture of the organization doesn't impact the performance of the work has not been keeping up with the literature [15, 19, 20 to cite a few].

The large-scale effort that is required to acquire, implement, and sustain organizational technology requires deliberate and well-coordinated strategic planning and implementation tasks in order to ensure effectiveness. However, both the technical and the business side of the organization often fail to understand the need to actively tailor those tasks to each organizational application. That lack of understanding is unfortunate since, without such a sustainable strategy it is highly unlikely that the employees of the organization will be properly motivated to produce artifacts at the requisite level of effectiveness and security.

Controls

The concept of business management controls might seem far removed from the issue of security flaws in software. But most of the activities that go into the production of a software system have to be overseen and controlled in some formal fashion [16]. Otherwise the organization runs the risk of important decisions being made at the lowest and most inappropriate levels of the organization [3].

Some form of formal governance control is necessary in order to ensure proper and adequate system assurance. Because of the high degree of skill and specialization required, details about software and systems are largely invisible to anybody above the basic technical level. The organization uses governance control mechanisms to influence and direct the way in which code is produced, maintained, and applied. While this has been the case since the beginning of the profession, the increased impact of technology within the functioning of the organization has greatly expanded the importance of these controls.

The problem is that, without specifically designed management controls, which are designed to enforce visibility, the evolution and even use of the system will take place without

the direct involvement of most of the organization, specifically all levels of management [15]. The inability to directly oversee technical work forces managers and the organization in general to rely on the capabilities, and even the good will, of employees who have no ability, or reason, to understand the overall strategic goals of the organization, including the desired level of software and system security.

Security controls are built into technical work in the same way that financial controls are built into the accounting process [16]. They must be intentionally designed discrete, systematic behaviors that can measure performance and then move the necessary information to the right decision maker at the right time. Controls allow the organization to both understand, as well as control the present status and long-term evolution of their systems. In the realm of software engineering the primary example of such controls would be the formally planned unit and integration tests and reviews that take place during development. Another example would be all of the formal steps taken to ensure proper configuration management [17].

The development and maintenance of systems as a whole has to be carefully coordinated in order to assure against the types of faults that are the basis for most of the exploits listed in the Common Weakness Enumeration (CWE). However, an effective governance system is also necessary to ensure that corrective action for all of identified defects is systematically initiated, overseen and then signed-off on. Thus, it can be shown that a rationally planned, implemented and executed governance control system is an important aspect of secure code.

Nevertheless the design and implementation of those controls is often left in the hands of business managers, who are no more knowledgeable about the software engineering process than software engineers are about financial accounting. To counteract this separation of knowledge, the organization as a whole has to make a concerted team effort to come up with meaningful and effective controls. This process does not take place by accident. It has to be planned and implemented as part of an overall software security effort. In that respect, control design and implementation is as much a part of the overall assurance process as static tests [21].

Reliance on incremental development places an even greater burden on these management decisions that directly impact operational security. Who will be making the determination as to when the operational security is sufficient to justify operational deployment and on what basis will they make their decisions? Planning for operational security must be included from the start [29].

Alignment

Proper business process to system alignment is a function of broad scale strategic management [21]. In essence, proper alignment ensures that all software and systems interact optimally with each other and all of the communities of interest that use them. The aim is to produce optimum performance and value for the organization [21, 22].

The aim of strategic alignment is to find the best top-down fit of all of the well-defined lifecycle primary, supporting, ancillary and management processes that are involved in the production and sustainment of software. Alignment is accomplished using

explicit process engineering methods best characterized by the ISO 12207-2008 model [26]. This is a very high level and concept based design exercise, with a concrete outcome in the form of a lifecycle infrastructure fitted to the specific environment of that particular organization.

Nevertheless, there are distinct payoffs for proper alignment in the system and software assurance universe. The need to maintain clear and robust linkages between systems ensures against attacks on the interface between systems and users. As a result, strategic alignment becomes a crucial issue in the maintenance of suitable software security. Consideration of alignment on the user interface can also ensure against social engineering scams. Those types of attacks are common methods for exploitation of gaps and misalignments in system operation [23].

The activities that ensure proper alignment are a function of two large software engineering processes, software quality assurance, which in this era also implies security assurance, and classic configuration management [24]. Those processes are well-defined in the Software Engineering Body of Knowledge (SWEBOK) and can be customized to any application aimed at maintaining monolithic alignment between the systems and software assets of any organization [25]. The ability to align all system and software assets in optimum operational harmony produces real outcomes. Those outcomes include attack surface reduction, gap assurance, and protection against the injection and over-run exploits that comprise the SANS top 25 list [27].

NIST in the recent release of the special publication NIST SP 800-160 Systems Security Engineering [30] defined alignment of security engineering with systems engineering and directly related the tasks of security to the engineering tasks described in ISO/IEC 15288 [31].

The problem comes from the fact that alignment is a strategic activity that can only be enforced at the top of the organization. The primary concern is that this process is rarely carried out in most businesses simply because C-Level executives see system and software alignment as "technical" work. Nonetheless, anything less than total alignment introduces the prospect of security vulnerabilities and is therefore likely to allow breaches that impact the overall mission of the organization. Thus upper level managers have to specifically authorize and delegate the responsibility for alignment in the same manner as the other major functions of the organization.

This is usually in the form of high-level technical managers with the authority to make strategic decisions about system development and deployment across the organization as a whole. It is necessary to focus that authority into a single coordinating entity in order to ensure uniform development of the larger system. It is also important to centralize authority for alignment into a single place for the purposes of oversight and enforcement.

Resourcing

The strategic business processes that most directly impact the security of systems and software are the resourcing decisions. A product is no better than the sum of the people who make it, the tools they utilize and the environment within which the work is performed. Accordingly, it is in the decisions that provide those resources that risks can be directly weighed and

evaluated and eventually either accepted, or mitigated [28].

The primary decision factors in resourcing far predate software and computers. Those are the classic elements of time and money. Decisions like schedule and delivery date impact the time available for assurance as well as the level and degree of inspection and testing. In the larger software engineering sense they also impact the amount of time that can be devoted to getting the specification and design right. Money dictates the number and types of people who can be devoted to a project. Funding impacts the tools available to verify designs and identify and remove defects. It also requires time to learn and apply tools effectively.

Staff capability impacts practically every aspect of the quality and security of the system and software portfolio. Nonetheless, individual staff capability is tied to resource decisions made in the larger sense. Those include such standard resource items as salary and staffing levels, which determines the type and level of talent available to a project. It also includes the general environment and the sophistication of the equipment that is utilized to do the actual work. Poorly staffed and supported software engineering teams are more likely to produce inferior and thus more insecure products.

But resourcing also embraces indirect factors such as whether to outsource. If outsourcing is the approach of choice, then resources determine how rigorously to monitor and control the contractors doing the work. Given the issues discussed earlier that are associated with supply chains, the level and degree of oversight and management of the outsourcing process can determine whether the products that are then integrated into the business operation are secure, or insecure.

Resourcing is often considered in the making of project plans. However it is not clear that resources are ever directly tied to assurance goals in those plans. That is because staff is often described in terms of the roles they fulfill rather than the criticality of the tasks. And the time allotted for project phases is most frequently dictated by the contract with the customer. Therefore it is important to also consider the sensitivity of the tasks themselves when considering how much money to devote to staffing the project. More importantly, it is critical to factor the assurance case into the business plan. That case is what should be a determinant for software engineering factors such as testing time, test sampling methods and most importantly of all the actual period of time that will be devoted to assurance.

Conclusions

More strategic, non-technical factors can have as great an impact on the security of the code as good programming practice. The decision makers involved in business strategy, organizational controls, business process alignment and strategic resourcing decisions must recognize the impact they have on operational security, understand the importance of coordinating their decisions with technology assurance needs, and accept responsibility for their choices. The strategic decisions affecting the processes, people, and tools should be thought about just as carefully and in as detailed a fashion as the specific software engineering tasks. That is not to suggest that we need to ignore secure coding advice and concentrate solely on strategy, alignment and resourcing. What this suggests is that the problem is a complex of small and

large factors, all of which have to be considered as a systematic entity in the assurance of systems and software.

Every one of the factors we discussed has concrete consequences in the real-world and therefore it is impractical to expect secure products without effective planning. Organizational context must be included when considering how to create a secure software engineering production and maintenance environment in order to achieve satisfactory assurance.

Disclaimers:

Copyright 2014 Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. ♦

REFERENCES

- Humphrey, W., *Managing the Software Process*, 1989, Addison-Wesley, Reading, MA
- Humphrey, W., *A Discipline for Software Engineering*, 1995, Addison-Wesley, Reading, MA
- Paulk, M. C., Weber, C. V., Curtis, W., Chrissis, M. B., "Capability Maturity Model for Software (Version 1.1)". Technical Report (Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University). CMU/SEI-93-TR-024 ESC-TR-93-177, <<http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=11955>>, accessed December 2014
- Curtis, B., Hefley, W.E., and Miller, S., *The People Capability Maturity Model: Guidelines for Improving the Workforce*, 2002, Addison Wesley Longman Reading, MA
- Ahern, D., Armstrong, J., Clouse, A., Ferguson, J., Hayes, W., and Nidiffer, K., *CMMI SCAMPI Distilled: Appraisals for Process Improvement*. Addison-Wesley Professional, 2005
- Software Engineering Institute, "CMMI for Development, Version 1.3" CMMI-DEV (Version 1.3), Carnegie Mellon University, 2010
- Software Engineering Institute, "CMMI for Acquisition, Version 1.3" CMMI-ACQ (Version 1.3), Carnegie Mellon University, 2010
- Software Engineering Institute, "CMMI for Services, Version 1.3" CMMI-SVC (Version 1.3), Carnegie Mellon University, 2010
- International Standards Organization, *ISO/IEC 15504, Information Technology Process Assessment (parts 2 through 9)*, ISO/IEC, Geneva, 2011
- International Standards Organization, *ISO/IEC 12207:2008 Systems and Software Engineering – Software Life Cycle Processes*, ISO/IEC, Geneva, 2008
- ResQsoft, "The Benefits and Disadvantages of Commercial Off The Shelf Application," <<http://www.resqsoft.com/benefits-disadvantages-commercial-shelf-application.html>>, accessed November 2014
- General Accounting Office, "Report to Congressional Committees - GAO High Risk Series," <<http://www.gao.gov/assets/660/652133.pdf>>, February 2013, accessed November 2014
- AcQnotes A Simple Source of DoD Acquisition Knowledge, *Software Management - Common Software Failure Causes*, <<http://acqnotes.com/acqnote/careerfields/common-software-failure-causes>>, accessed November 2014
- KPMG Consulting, "What went wrong? Unsuccessful Information Technology Projects," Toronto, Canada, <http://p3m.com.au/Reference/KPMG_1997.pdf>, accessed November 2014
- Charette, R. N., "Why Software Fails," <<http://spectrum.ieee.org/computing/software/why-software-fails>>, accessed November 2014
- Information Systems Audit and Control Association, "Framework for IT Governance and Control," ISACA, <<http://www.isaca.org/Knowledge-Center/COBIT/Pages/Overview.aspx>>, accessed November 2014
- ANSI/IEEE Std. 1042, *IEEE Guide to Software Configuration Management*, ANSI/IEEE Std 1042-1987
- The Open Web Application Security Project, "Command Injection Attacks," OWASP, <https://www.owasp.org/index.php/Command_Injection>, accessed November 2014
- Shahzad, F., Iqbal, Z., and Gulzar, M., "Impact of Organizational Culture on Employees Job Performance: An Empirical Study of Software Houses in Pakistan," *Journal of Business Studies Quarterly*, 2013, Volume 5, Number 2
- Grondahl, E., and Martinsson, L., "Impact of Organizational Culture on Quality Management," Chalmers University of Technology, Report No. E2011:025, Gothenburg, Sweden, 2011
- Van Grembergen, W., and De Haes, S., *Enterprise Governance of IT: Achieving Strategic Alignment and Value*, Springer, 2009
- Van Grembergen, W., and De Haes, S., "A Research Journey into Enterprise Governance of IT, Business/IT Alignment and Value Creation," *International Journal of IT/Business Alignment and Governance*, Vol. No. 1, 2010, pp. 1-13
- OWASP, "Top 10 2013 A1: Injection Flaws," OWASP, <https://www.owasp.org/index.php/Top_10_2013-A1-Injection>, accessed December 2013
- Meyer, A., "Configuration Management in the Security World," SANS Corporation, <<http://www.sans.edu/research/security-laboratory/article/meyer-config-manage>>, accessed November 2014
- IEEE Computer Society, "The Software Engineering Body of Knowledge 3.0," IEEE, <<http://www.computer.org/portal/web/webok>>, accessed November 2014
- Shoemaker, D. and Sigler, K., *Cybersecurity – Engineering a More Secure Software Organization*, 014Cengage Publishing, 2013
- SANS, "CWE/SANS Top 25 Most Dangerous Software Errors," SANS Corporation, <<http://www.sans.org/top25-software-errors/>>, accessed November 2014
- McGraw, G., *The Role of Architectural Risk Analysis in Software Security*, Pearson, 2006
- Woody, C., "Agile Security – Review of Current Research and Pilot Usage" Carnegie Mellon University Software Engineering Institute, April 2013, <https://resources.sei.cmu.edu/asset_files/whitepaper/2013_019_001_70236.pdf>, accessed November 2014.
- National Institute of Standards and Technology, "DRAFT Systems Security Engineering: An Integrated Approach to Building Trustworthy Resilient System," NIST SP 800-160, May 12 2014, <http://csrc.nist.gov/publications/drafts/800-160/sp800_160_draft.pdf> accessed November 2014
- International Standards Organization, "Systems and software engineering – System life cycle processes", ISO/IEC 15288:2008, ISO/IEC, Geneva, 2008

ABOUT THE AUTHORS



Daniel P. Shoemaker, PhD, Principal Investigator and Senior Research Scientist at UDM's Center for Cyber Security and Intelligence Studies. This Center includes the Computer Information Systems-Information Assurance Department, as well as the Center of Academic Excellence for National Security Agency. The Center has just completed a two-year Department of Defense Contract to develop Software Assurance Curriculum and Courseware. Dan is a full time Professor at University of Detroit Mercy with 25 of those years as Department Chair. As the Co-Chair for the, National Workforce Training and Education Initiative he is one of the Authors of the National Software Assurance Common Body of Knowledge (CBK) for the Department of Homeland Security. But while Dan spends a lot of time in DC, he is a Michigan man at heart, beginning with his education at the University of Michigan and the outreach opportunities he shepherds within the State of Michigan through his leadership of the International Cyber-Security Education Coalition. This Coalition covers a five state region with research partners as far away as the United Kingdom. Dan also spends his free time authoring some of the leading book in Cyber Security. Look for his newest edition to hit the press, this spring *Cyber Security : The Essential Body of Knowledge*, based on the DHS National Cyber Security Division's EBOK. His last book, *Information Assurance for the Enterprise*, is McGraw-Hill's primary textbook in that field and was number one on Amazon's list for three consecutive years. His next book *Engineering a More Secure Software Organization*, which is also published by Cengage, will be out next spring.

Phone: 313-680-1434

E-mail: dan.shoemaker@att.net



Dr. Carol Woody has been a senior member of the technical staff at the Software Engineering Institute, Carnegie Mellon University since 2001. Currently she is the technical lead of the cyber security engineering team whose research focuses on building capabilities in defining, acquiring, developing, measuring, managing, and sustaining secure software for highly complex networked systems as well as systems of systems.

Phone: 412-268-9137

E-mail: cwoody@cert.org

**CIVILIAN TALENT IS MISSION-CRITICAL.
LET'S GET TO WORK.**

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to www.navair.navy.mil.

Equal Opportunity Employer | U.S. Citizenship Required

**NAVAIR
CIVILIAN**

CHOICE IS YOURS.

What is a Software Engineer?

Al Kaniss, Naval Air Systems Command Headquarters

Many years ago, at a Software Technology Conference, one of the briefers talked about how creating computer code had gradually evolved from art to craft to engineering. The briefier had a slide with a graphic of a doghouse, a typical suburban single-family house, and a skyscraper. His point was that the difference between the three structures wasn't just the progressively larger size, but rather the vastly increasing complexity. The briefier explained that your teenager could probably successfully build a doghouse, but couldn't build the family house. And the builder who built your family's house most likely couldn't build a skyscraper. Software Engineering is exponentially more complex than cobbling code together which satisfies a functional requirement.

That mental model of size versus complexity stuck with me ever since. In fact, I use a similar graphic (figure 1) three times a year, when I brief new NAVAIR Systems Engineers about software and software engineering. People still tend to equate software engineering with computer programming. While programming, i.e., writing computer code, is certainly an important facet of the software engineer's duties, it may be as little as 20 to 30 percent of those duties.

Yes, in the early days of software development, things were vastly different. Computers didn't have much memory, so programs couldn't be very large. Computers weren't networked nor even accessible by the average person. They were either in large, locked rooms tended to by operators in white lab coats, or in specialized labs associated with weapons systems. Just entering a small program by setting switches, observing lights and getting it to run successfully was a major accomplishment.

Contrast that with today's environment. The majority of people on the planet have access to a fairly powerful computer with access to most of the other computers on the planet via the Internet. Cybersecurity, long ago not much more than preventing access to a locked room and changing one's password periodically, has become one of the major responsibilities of a software engineer.

Along with increased complexity, software development also got more disciplined. Back in the early days, a software developer just kept typing code and running it until the desired result was achieved. For small, simple computer programs, this was time consuming but adequate. As computer capacity grew and grew, and teams of developers replaced the lone computer programmer, more discipline was required. Requirements, designs, integration plans and tests had to be formally generated, documented, tracked and traced to assure that the whole team was in sync with a high probability of an integrated, successful product, especially as we've gone from computer programs of hundreds of lines of code, to suites of software for systems of over 24 million lines of code. SEI's CMM® and the CMMI® were created out of necessity for increasing the discipline of software development teams and became commonly used by organizations large and small.

The goal of those involved with software development has never changed: creating a reliable, functional computer program. The responsibilities of people who are involved with doing so, however, have grown enormously. Software Engineers are fully engaged with Systems Engineers to decompose system requirements into software requirements. Software architecture and design have become increasingly important, especially as systems have been networked and have become "systems of systems". The tasks of software integration and system integration have grown exponentially as the number of computer modules (CSCIs, CSCs and CSUs) and subsystems grows with computer capacity.

Software safety has also become an increasingly significant responsibility of Software Engineers over the past 25 years, as software has been given increasing control over systems. As mentioned, Cybersecurity is also a major concern these days. Regardless of how well software functions, if it is not protected from its threats, its value diminishes greatly.

Software Engineers must also create and maintain software that satisfies a lot of other demands. Since the life of software can be 10, 20, 30 or more years, it must be designed to be easily re-hosted on newer hardware without major (and thus costly) changes. Software must also be easily modifiable over its life. We've all heard of "spaghetti code", which can be more difficult and costly to modify than it was to create, especially if the people maintaining the code had not written it in the first place or lack adequate documentation detailing it.

Another attribute that people demand of software these days is that it be reliable. This can cause a lot of confusion as software is always reliable, in that it doesn't break, wear out or rust out like hardware does. The software however operates within a system, which is really the entity that must be reliable. The state of the computer (including other software executing and operator and other external inputs) can make the software appear unreliable. It is up to the Software Engineer to design the software to be tolerant of such things.

Users of computer software want it to be "user friendly". User friendliness is of course an ambiguous requirement, and if you ask

Software Engineering: More Than Just Writing Code

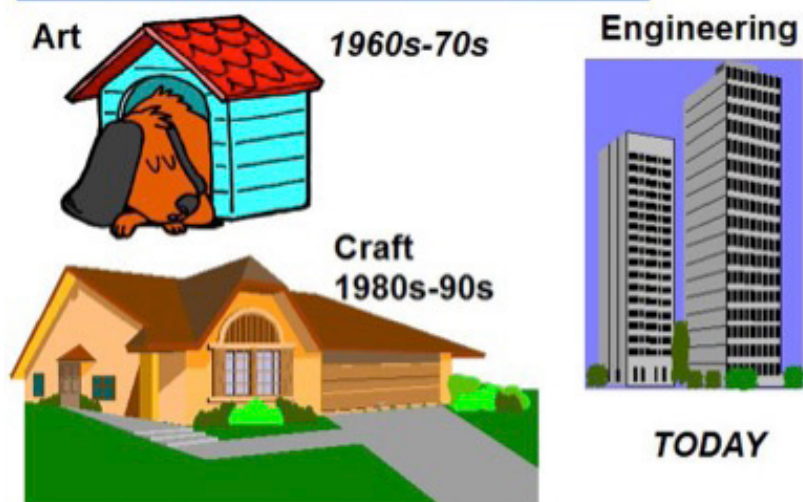


Figure 1

100 users how they want some piece of software to look and behave, you will likely get 100 different answers. And likewise, as users get more familiar with the software, they want the user interface to grow from “beginner mode” to “expert mode”, with fewer prompts and more complex screens as their expertise grows.

Complicating everything else, there is more and more pressure to field software more quickly and thus more cheaply, requiring Software Engineers to increasingly learn and use Agile methods. We want software that is of high quality, produced quickly, and at minimal cost. That is quite a tall order. Add to those other attributes we require of software (figure 2), the Software Engineer has the Herculean task of satisfying all the people all the time. And often, such attributes conflict. For example, making a system “open” to decrease costs and facilitate software re-use conflicts with making a system secure.

The explosive growth of software-reliant systems vastly increases the need for talented software engineers. Hopefully, as time goes on, we will continue to develop enough people who have the full complement of skills necessary to accomplish such work and attract them to work in the Defense environment.

Ironically, there is no Software Engineer title in the federal government. We hire people into the existing Computer Engineer, Electronics Engineer and Computer Scientist billets. Hopefully someday soon, such a title will exist. It's also ironic

that in the early days of Software Engineering, one of the newer engineering fields, some of the more traditional types of engineers (civil, mechanical, electrical) tended to challenge the notion of a “software engineer”, since it didn't involve physical things like buildings and bridges. Hopefully, that opinion is long past as software has become so critical a component of virtually every system that is produced today.

Disclaimer:

CMMI® and CMM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. ♦

ABOUT THE AUTHOR



Al Kaniss has worked for the Navy for 39 years in various capacities as a Software Engineer...long before “Software Engineer” was even a common term. He is now Branch Head for Software Engineering at Patuxent River Maryland, home of Naval Air Systems Command Headquarters.

Category	Quality attribute	Description
Design Qualities	<i>Conceptual Integrity</i>	Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed, as well as factors such as coding style and variable naming.
	<i>Maintainability</i>	Maintainability is the ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the functionality, fixing errors, and meeting new business requirements.
	<i>Reusability</i>	Reusability defines the capability for components and subsystems to be suitable for use in other applications and in other scenarios. Reusability minimizes the duplication of components and also the implementation time.
Run-time Qualities	<i>Availability</i>	Availability defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period. Availability will be affected by system errors, infrastructure problems, malicious attacks, and system load.
	<i>Interoperability</i>	Interoperability is the ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties. An interoperable system makes it easier to exchange and reuse information internally as well as externally.
	<i>Manageability</i>	Manageability defines how easy it is for system administrators to manage the application, usually through sufficient and useful instrumentation exposed for use in monitoring systems and for debugging and performance tuning.
	<i>Performance</i>	Performance is an indication of the responsiveness of a system to execute any action within a given time interval. It can be measured in terms of latency or throughput. Latency is the time taken to respond to any event. Throughput is the number of events that take place within a given amount of time.
	<i>Reliability</i>	Reliability is the ability of a system to remain operational over time. Reliability is measured as the probability that a system will not fail to perform its intended functions over a specified time interval.
	<i>Scalability</i>	Scalability is ability of a system to either handle increases in load without impact on the performance of the system, or the ability to be readily enlarged.
	<i>Security</i>	Security is the capability of a system to prevent malicious or accidental actions outside of the designed usage, and to prevent disclosure or loss of information. A secure system aims to protect assets and prevent unauthorized modification of information.
System Qualities	<i>Supportability</i>	Supportability is the ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
	<i>Testability</i>	Testability is a measure of how easy it is to create test criteria for the system and its components, and to execute these tests in order to determine if the criteria are met. Good testability makes it more likely that faults in a system can be isolated in a timely and effective manner.
User Qualities	<i>Usability</i>	Usability defines how well the application meets the requirements of the user and consumer by being intuitive, easy to localize and globalize, providing good access for disabled users, and resulting in a good overall user experience.

Figure 2 – Software Quality Attributes

Is Software Engineering Really Engineering?

Lawrence Peters, Software Project Management Consultant

Abstract. In an attempt to improve the cost and quality issues in software, more than 40 years ago, NATO sponsored a conference which may have formally introduced the term, "Software Engineering" to the world. The term held out hope that the software community would improve matters if they viewed what they were doing as engineering rather than some form of cottage industry based on computer code. The fact was that what engineering meant as a profession was unfamiliar to most software developers at that time and still is a serious challenge to today's software professionals.

At the time of the first NATO conference [1] on software engineering, people from a broad range of backgrounds had found a home in software development. So much so that studies were done to find out just what it was about software development that drew in people from such a diverse set of backgrounds [2]. What they found was a predominant psychological profile consisting of a pair of attributes that were unique when taken together. Of the 60 professions studied, software developers were unique in that they had:

- **High GNS (Growth Need Strength)** – This represents a strong desire to solve significant, difficult problems, particularly problems which were at the leading edge of technology or moved the technology frontier forward. If you have ever wondered why it is so difficult to get software engineers to document their work, it could be due to this trait.

- **Low SNS (Social Need Strength)** – This represents a strong desire to work alone rather than as a contributing member of a team. Most upgrades to existing software systems are complex enough that a team of people are required to accomplish them. This trait makes building that team more difficult than one might expect.

These characteristics were present regardless of the discipline these folks were originally trained in. For example, the top 10 software developers I have ever worked with over the last 40 years include a high school dropout, an M.S. in Psychology, an M.S. in Library Science, an Art History major, a mathematician, a theoretical nuclear physicist and others (including a couple of people with engineering degrees) from a total of 5 different countries. Granted, this is only one person's experience but those who have been in the software field for more

than 20 years probably have had a similar experience. At some time during their academic or professional careers, people with this profile get exposed to computer programming and become "hooked" abandoning their original vision of how they would make their living in this world for what we now view as software engineering. Another attractant of software engineering is the fact that the work one does can be highly personalized, expressing one's own "style" and not having to conform to some rigid rules written by others. Oddly enough, this is in spite of the fact that the syntax and semantics of the programming language one uses are not open for discussion but they set the bounds within which a program can be created.

Today, the population in software development appears to be much more uniform perhaps due to the popularity of college majors like computer science, information systems and others including, software engineering. At this point it is fair to ask, "What is engineering?" The international organization tasked with defining and maintaining definition of the term engineering (ABET Accreditation Board for Engineering and Technology) defines engineering as, "the profession in which knowledge of the mathematical and natural sciences gained by study, experience, and practice is applied with judgment to develop ways to utilize economically the materials and forces of nature for the benefit of mankind." Under that definition, software engineering is a "rather odd bird." In the more traditional branches of engineering, we are dealing with the forces of nature (e.g. physics, chemistry) which continue to work as they always have or at least, we think so. Software is definitely different in that nearly everything we deal with during even relatively simple projects changes – the requirements, budget, schedule, user interface all undergo nearly continuous evolution over the course of the project.

At this point, we can conclude that software engineering is not really engineering at all or it is the prototype of a new form of engineering dealing in the realm of logic as it is applied to functionality to benefit mankind. Let's take a look at each of these viewpoints:

- **Software engineering is not engineering** – A less restrictive definition of engineering is, "the process of developing a system that conforms to accepted engineering and development practices meeting or exceeding functional, performance and reliability requirements." Under this one, software engineering qualifies as engineering provided the term system includes software.

- **Software engineering is engineering** – Under the previous definition, software engineering survives classification as engineering. But it also has adopted many of the accoutrements of the more traditional engineering disciplines. These include the development and refinement of a body of knowledge (SWEBOK) [3], standards for various activities comprising the software development process and conferences and research in various parts of the world focused on specific topics of concern to software professionals.

A concern shared by many software engineering professionals is the seeming inability of our profession to meet budget, schedule and quality issues. This appears to be a symptom of our parochialism because other, well-established branches of engineering miss their targets with a great deal of consistency [4].

Figure 1:

Type of Project World Wide	Average % Overrun	Overrun % Range
Railways	45	7 to 83
Bridges & Tunnels	34	-28 to 96
Roads	20	-10 to 50

It turns out our seeming inability to estimate with a fair amount of accuracy is actually a characteristic shared by all human beings [5]. Identifying it resulted in a Nobel prize in Economics in 2002. He demonstrated that everyone misses estimates due to 2 factors:

1. An over estimation of our abilities – Just about every software project manager has sat in a meeting with the development team to discuss the cost and schedule issues related to a proposed change and have the team indicate the change would be a “piece of cake” only to find that when the change is actually put into place it is a cost and schedule nightmare.

2. An over estimation of the benefits of the effort to the near exclusion of consideration of risks – In combination with #1, this factor is particularly dangerous when submitting a bid for a contract in competition with other firms. The team looks at how much they can make on the contract, play down risks that can cause overruns and ignore the complexities inherent in the proposed effort.

Since these are traits common to us all, no branch of any profession has mastered accurately predicting the future. However, there is a way to back out this over optimism [6] changing the estimating problem. It involves a process whose conclusion results in the cost estimate being tempered by an x-y graph of dollar amount versus confidence level. The dollar amount represents the size of the set aside (also referred to as a “contingency fund”). The higher the dollar amount, the higher the confidence level that the project will be completed within the budget + contingency amount. The slope of the line used to identify the contingency fund level. The data used to compute this chart is based on your team's experience in estimating and executing similar projects. It is so effective that the American Planning Association has recommended its use over traditional planning and estimating methods [4].

But the key issues in all of this remain:

- What is engineering?
- Does software engineering qualify as engineering?

Dictionary definitions of engineering provide a relatively straightforward benchmark for us. According to Webster's dictionary [7] engineering is defined as: “The design and manufacture of complex products”

The Problem

No matter what facet of software engineering people work in, they are reluctant to identify themselves as a, “Software Engineer.” This may be due to some social image of engineers as being boring or old fashioned or something else unfashionable. Perhaps this will change as the public's awareness that software engineers make a decent living and work in a generally low hazard environment. A part of the problem we have not addressed is the issue of ideology or belief system. What people believe or accept as fact is extremely difficult to change. Dissuading a belief involves a lot more than facts and data [8]. A crisis

or some other catastrophic event may accompany a change but this is not always certain [8]. It all depends how we attribute the failure or misinformation – to ourselves or some other factor [9].

Regardless of what you define as engineering, all engineers work in an environment marked by:

- There is a problem to be solved
 - Solving the problem is complex because it entails solving many subsidiary problems
 - The solution will benefit one or more companies, governments, or members of society
 - Arriving at the problem solution must be bounded by the budget and timeframe allotted
 - There is no guarantee that the problem can be solved
- Also, engineers generally follow a common process which generally proceeds as follows:

1. There is a problem to be solved
2. The problem and its proposed solution are documented in a way that both the client and the engineering team can understand
3. A design is proposed and cost estimate developed
4. If approved by the client, implementation of the design proceeds via a project plan, costs detailed and prototypes and/or mockups developed together with acceptance criteria and so forth.
5. Upon completion and compliance with the acceptance criteria, the system is delivered

As we can see from this over simplified depiction of the engineering process, all engineering processes share strong similarities. The IEEE has made the some similarities even stronger via the creation and maintenance of a series of standards and the Software Engineering Body of Knowledge [3]. In addition, the Software Engineering Institute developed the Capability Maturity Model and its successors consistent with ISO 15504 and ISO's efforts worldwide to assist organizations improve the quality of engineered products.

What Can We do to Promote the Use of the Term Software Engineer?

If you take a look at the schools that offer software engineering as a major at the graduate or undergraduate level you will find these programs are run under the auspices of the computer science department, the information systems department and other department titles – not the school of engineering. Why? I am not sure but more than 30 years ago, in authoring the first graduate level curriculum in software engineering at Seattle University, we made sure it was part of the school of engineering. Students entering that program knew right from the start that this was an engineering discipline and, if successful in it, their degree would be awarded by the engineering department. What has happened since then in more than 100 universities around the world is the creation of software engineering programs mostly run by the computer science department [10]. An examination of these curricula reveals that the adage, “We teach what we know” still holds true. The programs are largely about programming methods, general condemnation of the Waterfall lifecycle and often sprinkled with esoteric topics related to code production. Even after all these years, the United States Bureau of Labor Statistics in its latest update [11] online lists many categories of engineer and engineering but does not list soft-

ware engineer as a labor category. Perhaps it is a lack of self-promotion or the comparative newness of this discipline which has prevented software engineering from being recognized as a legitimate branch of engineering. For example, civil engineering has been around since before the time of the ancient Romans.

Is There an Alternative?

If we did not call this profession Software Engineering, what would we call it? Alternatives do not readily come to mind especially since this label has been around for so long. Perhaps the best course of action is for us to gain more knowledge of other engineering fields and to identify ourselves as engineers. This may go a long way to making Software Engineering a permanent part of the engineering lexicon.

Disclaimer:

Copyright © 2015 Software Consultants International Limited. All rights reserved. ♦

REFERENCES

1. Naur, P. and Randell, B. (Editors), "Software Engineering: Report on a Conference Sponsored by the NATO Science Committee," Garmisch, Germany, 7th to 11th October, 1968, Brussels, Scientific Affairs Division, NATO, January, 1969.
2. Cougar, D. and Zawacki, R., "Motivating and Managing Computer Personnel," Wiley-Interscience, New York, N.Y., 1980
3. IEEE, "Guide to the Software Engineering Body of Knowledge," Version 3.0, available for download, December, 2014.
4. Flyvberg, B., "From Nobel Prize to Project Management: Getting Risks Right," Project Management Journal, August, 2006, pp. 5 – 15
5. Kahneman, D., 1994, New challenges to the rationality assumption. Journal of Institutional and Theoretical Economics, 150, pp 18-36
6. Peters, L. J., "Managing Software Projects: On the Edge of Chaos – From Antipatterns to Success," Kindle ebook, Software Consultants International Limited, 2015.
7. Merriam-Webster on line dictionary, December, 2014.
8. Festinger, L., Riecken, H. and Schacter, S., "When Prophecy Fails," Martino Fine Books, Eastford, Connecticut, 2012
9. Myers, C., Staats, B. and Gino, F., "My Bad! How Internal Attribution and Ambiguity of Responsibility Affect Learning from Failure," Harvard Business School Working Paper 14-104, April, 2014.
10. Peters, L. J., Unpublished Preliminary Results of Internet Search in Support of Software Project Management Course as part of EMSE Program, Universidad Politecnica de Madrid, 2014.
11. United States Department of Labor website job categories listing, latest update, January 8, 2014.

ABOUT THE AUTHOR



Dr. Peters has four decades experience in software engineering as a software engineer, project manager, consultant, and educator (he currently teaches Software Project Management in Spain via Skype). He has worked on many defense systems. His clients have included many Fortune 100 companies, the US Department of Defense, and the Canadian Defence Establishment. He has a B.S. in Physics, an M.S. in Engineering and a PhD in Engineering Management. He created the first Software Engineering laboratory for the Canadian Air Force, written 4 books and published several papers

E-mail: ljpeters42@gmail.com



CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, **CROSSTALK** can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for the areas of emphasis we are looking for:

Supply Chain Assurance

Sep/Oct 2015 Issue

Submission Deadline: Apr 10, 2015

Fusing IT and Real-Time Tactical

Nov/Dec 2015 Issue

Submission Deadline: Jun 10, 2015

Please follow the Author Guidelines for **CROSSTALK**, available on the Internet at www.crosstalkonline.org/submission-guidelines. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit www.crosstalkonline.org/theme-calendar.

SISE: A Novel Approach to Individual Effort Estimation

Russell Thackston, Georgia Southern University
David Umphress, Auburn University

Abstract. The SISE estimation model presents a novel alternative to PSP's PROBE by combining expert judgment with empirical data to create reasonably accurate predictions. The model uses a simple, four-steps process in which a future activity is compared to a developer's completed tasks to identify two known effort values that define a prediction interval. Initial validation of the SISE model by researchers at Auburn University was completed in July 2013 and development of supporting tools is currently underway.

Introduction

In recent years, dramatic changes to the software industry have brought individual developers to the forefront of software engineering practices. In addition, the rise of the software micropreneur in markets such as mobile app development and web applications has reinforced the need for lightweight, agile software engineering practices focused on individuals as opposed to teams. For example, recent surveys of the microISV industry have shown that time management and related issues topped their founders' list of "pain points"¹. Historically, however, many of the software process tools available to software engineers have been team-oriented, making them impractical for the individual to benefit from their usage².

In response, researchers at Auburn University have been focusing their efforts on constructing tools targeted directly at the individual software engineer. One such tool is the SISE effort estimation model.

SISE is a lightweight, agile model designed to construct estimates based on expert knowledge and empirical evidence. In this respect, SISE outperforms simple guesswork, while incurring a much lower overhead than traditional, established models – such as the Personal Software Process (PSP) PROxy-Based Estimation (PROBE) model – which rely on complex software, algorithms, or mathematical calculations.

Traditional Models

A variety of established models exist for estimating the size of a future task. The vast majority – Planning Poker, Wideband Delphi, COCOMO, etc. – focus on team- or project-level estimation. Most of these models have demonstrated an ability to accurately predict future effort through the combined efforts of a team. However, the individual software engineer is often left to his or her own devices when it comes to planning personal activities on a daily basis.

Currently, only one formal estimation model exists that is specifically targeted at the individual software engineer: PSP's PROBE³. The PROBE model relies on a proxy-based approach in which estimators:

- Develop their conceptual design;
- Identify and size proxies for the actual work;
- Estimate other elements;
- Estimate the program size using one of four methods;
- Calculate prediction intervals.

PROBE forecasts effort through a set of rules that determine the statistical relationship between size estimates and actual effort across past projects. The approach falls back on "engineering judgment" when no demonstrable relationship exists.

The disadvantage to PROBE is in its perceived complexity and inflexibility. The rules for determining which pieces of historical data provide the strongest statistical mapping of size to effort niggles developers, especially in the absence of any tool. The approach requires a substantial amount of bookkeeping. It relies on statistically significant amounts of historical data from relatively stable development efforts to avoid consistent use of the "engineering judgment" rule. On the plus side, PROBE does have the advantage of being the sole prescriptive model for costing effort at the level of the individual engineer.

When pressed for a quick, familiar approach to estimation -- especially in informal settings -- individual developers rely on expert judgment. While expert judgment is recognized as a valid and sometime accurate approach to effort estimation, it lacks formal, quantitative methods. Formal methods that may be incorporated into expert judgment include estimation by analogy, case-based estimation, and work breakdown, but the greatest challenge to expert judgment is in what to change to improve accuracy if the approach leading to the estimate is undefined or vague.

The SISE Model

"SISE" is an acronym for the model's four-step process. The four steps, in order, are Sort, Identify, Size, and Evaluate. The first step – Sort – involves the ordering of historical data by the actual effort required to complete the activity. The second step – Identify – involves choosing two tasks from the historical data set: one confidently known to be smaller, one confidently known to be larger, and both relatively close in size to the future work. Using this pair of tasks, the estimator begins the third step – Size – by producing a rough prediction interval of the future activity's size using the actual effort values for the two completed tasks. The final step – Evaluate – involves shifting or resizing the prediction interval to account for any detectable historical bias. This last step is optional and is only applied if the estimator is dissatisfied with the precision, accuracy, or confidence level of his or her estimate.

The design of the SISE model focuses specifically on the individual software engineer. Its estimates are based solidly on empirical data gathered by the software engineer and only applicable to that person. Personal skills and experiences are too numerous to list, quantify, and apply to every estimation scenario. Therefore, the SISE model seeks to join empirical data to the process of expert judgment. This results in a model that must be individually calibrated by each software engineer using his or her own personal data.

The goal of the SISE model is to provide a viable alternative to traditional models such as PROBE. The SISE model defines an agile approach to estimation by offering a lighter-weight method than those found in models such as PROBE and PCSE; the model employs fewer steps, which are demonstrably less complex. Additionally, the SISE model improves upon simple guesswork by relying on a foundation of empirical data.

The SISE Steps

Step 1: Sort

The Sort step involves the ordering of historical data by the actual effort required to complete the activity. The simplest approach is to maintain an electronic record of historical data, such as a spreadsheet or database. The data is then sorted by the actual effort, from smallest to largest. Next, the numeric values associated with each historical data point – estimated effort, actual effort, etc. – are hidden, leaving only the text description of the completed tasks; this prevents the software engineer from selecting tasks based on a desired numeric outcome, such as “eight hours.”

Step 2: Identify

Next, the description of the future activity is compared to the descriptions of the historical tasks. Two historical tasks must be located: one confidently smaller and one confidently larger than the future activity. The smaller task should be one which the estimator is confident is smaller than the future activity, but is as close as possible in size to the future activity. The larger task should be the inverse: larger in size, but still relatively close. Since the historical data set is already sorted, a very efficient way of locating these two tasks is through the use of a binary search algorithm.

The exact manner in which tasks are chosen as smaller and larger is left to the practitioner, as is the determination of confidence. In this instance, SISE views the practitioner's decision-making process as a black box; the model relies on the software engineer's intuition and experience to make complex, yet relatively accurate, judgment calls. The only other important consideration is consistency from estimate to estimate.

Step 3: Size

Once the practitioner has chosen a pair of tasks, the third step – Size – produces a rough prediction interval of the future activity's size. The size of the future activity is inferred by looking at the actual effort values of the two historical tasks. For example, assume the historical record contains twenty completed tasks and the estimator has selected tasks 9 and 14 as the two tasks confidently believed to be smaller and larger, respectively, than the future activity. The rough size of the future activity can, therefore, be inferred to fall between the actual sizes of tasks 9 and 14.

Prediction intervals are expressed using a low estimate and a high estimate, with the actual value expected to fall somewhere in between. Prediction intervals are expressed using the notation “[low, high].” For example, the prediction interval [5, 7] means we expect the actual value to fall somewhere between five and seven hours, inclusive⁴.

The actual effort values associated with the bracketing tasks represent the low bound and high bound of a prediction interval. However, this interval is a rough estimate of the expected effort and may need to be refined.

Step 4: Evaluate

The final step – Evaluate – is optional and may be applied in the event the estimator is dissatisfied with the precision, accuracy, or confidence level of the estimate. The estimator may choose to shift the prediction interval based on an analysis of his or her historical bias. This involves analyzing the practitioner's track record with using SISE and determining if the estimates are typically low or high. If a consistent bias can be demonstrated, then the future estimate is adjusted to account for the bias. The Removing Shift Bias section describes a sample scenario.

If adjusting for shift bias is not possible or has not produced the desired results, the practitioner may adjust for width bias. Width bias results from historical prediction intervals that are too narrow or too wide. Large prediction intervals create a high level of confidence, but make the estimate less useful for planning purposes through reduced precision. By comparing a future task's prediction interval width and confidence level to those of historical tasks, the practitioner may choose to increase or decrease the prediction interval width, and, therefore, achieve a more desirable precision/confidence level for the future task's estimate. The Removing Width Bias section describes a sample scenario.

This last step implies a prerequisite: the practitioner has been using SISE or some other prediction interval-based estimation approach and has an idea of his or her historical accuracy. This historical bias is then used to modify the rough estimate to produce a specific estimate.

It should be noted that within the SISE model estimation bias is not an indication or measure of error committed by an estimator. Rather, it is a measure of how the best efforts of the estimator translate through the SISE model to create an estimate that mirrors actual effort.

Framework vs. Methodology

The SISE model outlines a general framework for constructing estimates. However, the model is not prescriptive in terms of the lower level details of the process. The manner in which a software engineer tracks his or her time, identifies the smaller and larger tasks, and determines an acceptable prediction interval width/confidence level, are left to the practitioner. In these instances, the only important consideration is consistency; time tracking and relative task sizing must follow a similar method from day-to-day and week-to-week.

Additionally, the SISE model allows the practitioner to select an appropriate level of granularity for his or her work tasks. Typically, a software engineer must plan each day's activities. SISE facilitates such planning and, indirectly, supports longer term planning (e.g. weeks and months), by providing a quantitative estimate at

the lowest levels. Therefore, individual software engineers must determine the smallest unit of work that is appropriate for them personally for planning purposes. These units of work will define scope and frequency of each SISE estimate. Again, the most important aspect is consistency; the granularity of estimates should not change dramatically from week-to-week or month-to-month.

Getting Started with SISE

Introducing the SISE model into an individual's software process is simple. As with all regression-based approaches, the first step is to begin tracking effort expended to complete the work activities. As each new task is completed, it is recorded in the historical record with its description, estimated effort, actual effort, etc. This historical record will be the basis for all future estimates. If an estimator has already been tracking his or her time, then this information may be used, as long as it matches the granularity of the future activities to be estimated.

The software engineer produces a SISE estimate by reviewing his or her historical record. The historical record is sorted from smallest to largest by actual effort and the numeric values are hidden from view (step 1). The engineer reviews the list looking for a task that he or she is confident is smaller than the future activity. If a task is located (step 2), the actual effort is revealed and that value is recorded as the low end of the future task's prediction interval (step 3). If the estimator is not confident that any historical task is smaller than the future activity, then a value of zero is recorded as the low end of the future task's prediction interval.

Next, the estimator reviews the list a second time to locate a confidently larger task, again using only the descriptions of the future and historical tasks. If one is located, the actual effort value is revealed and recorded as the high bound of the future task's prediction interval. If a larger task cannot be confidently identified, then the upper bound of the future activity's prediction interval is recorded as "unknown" using the sign for infinity (∞).

With the prediction interval for the future activity established, the software engineer proceeds with work on the activity. Once the activity is completed, the actual effort is recorded in the historical record and the process repeats.

Accuracy, Precision, and Confidence Level

By using a prediction interval as the basis for estimates, the SISE model presents the software engineer with a competing set of factors: accuracy, precision, and confidence.

The accuracy of an estimate is measured in different ways depending on the type of estimate. Many project managers and project management applications expect an effort estimate to be phrased as a single value. Single value estimates are easy to understand, simple to aggregate, and are expected to be wrong. After all, what is the probability that an activity estimated at 10 hours will take exactly 600.00 minutes? Therefore, the accuracy of a single value estimate is measured in terms of its error (see section titled Measuring Accuracy).

The accuracy of a prediction interval, on the other hand, is measured by how often the actual effort falls within the interval. The overall percentage of actual effort values falling within their prediction intervals is known as the hit rate. Several logical observations can be made about the use of a hit rate. First, wider

prediction intervals are less precise and will typically produce higher hit rates; conversely, smaller prediction intervals are more precise and will typically produce lower hit rates. In other words, precision and accuracy are inversely proportional, generally tasking the estimator with balancing the two.

For ease of use, the SISE model deliberately takes a statistically simplistic approach to assigning confidence levels; the model assumes the software engineer will repeatedly employ the same method for determining relative size and creating estimates. Based on this assumption, the estimator's past performance can be used as a predictor of future performance. For example, if an estimator's hit rate is 50%, it can be said that half of the activities they have estimated have had actual effort values that fell within his or her prediction interval. Therefore, all things being equal, a new estimate has a 50% probability of being correct. Put another way, the estimator has a 50% confidence level in his or her next estimate.

Note that confidence level should not be confused with an estimator's logical or emotional confidence in his or her abilities and estimates. It can be assumed that when an estimator produces an estimate, he or she does so to the best of their ability; the estimator is confident the estimate is correct. Confidence level, on the other hand, is a measure of the probability that the estimate will be correct and allows the estimator to make statements such as:

In the past, my estimates have been correct 90% of the time. Therefore, I have a 90% confidence level in my next estimate, which I feel confident I have done my best in constructing.

Beginning with the first estimate, the SISE model assigns each new estimate a confidence level based on the estimator's current hit rate. As noted in the fourth step of SISE, however, the estimator may take steps to adjust this confidence level by compensating for historical bias (see the Adjusting for Width Bias and Adjusting for Shift Bias sections). Note that shift and width biases are not to be viewed as errors on the part of the estimator; rather they are to be viewed as the manner in which the SISE model adapts to an individual software engineer's perspective of past and future work.

SISE Example

Assume a software engineer, who has never engaged in time tracking, has decided to begin using the SISE model for his web development project. The developer been assigned a new work activity: "Design security model." Given that the software engineer's historical record is empty, he has no data points for an estimate; no smaller task or larger task can be identified to use as the basis for a prediction interval. Therefore, following the SISE model, the prediction interval for the first activity is $[0, \infty]$. Once the first activity is completed and the actual value is recorded, the hit rate is calculated to be 100% (see Table 1).

Task	Low Est. (hours)	High Est. (hours)	Actual (hours)
Design security model	0	∞	10

Table 1: One completed activity (Hit rate = 100%)

Task	Low Est. (hours)	High Est. (hours)	Actual (hours)
Design user model	0	10	8
Design security model	0	∞	10

Table 2: Two completed activities (Hit rate = 100%)

Task	Low Est. (hours)	High Est. (hours)	Actual (hours)
Design user model	0	10	8
Design security model	0	∞	10
Design content model	8	10	11

Table 3: Three completed activities (Hit rate = 67%)

Task	Low Est. (hours)	High Est. (hours)	Actual (hours)
Design database tables	0	11	6
Design user model	0	10	8
Design security model	0	∞	10
Design content model	8	10	11

Table 4: Four completed activities (Hit rate = 75%)

Task	Low Est. (hours)	High Est. (hours)	Actual (hours)	Missed Prediction Interval?
Design FAQ model	0	4	2	
Create FAQ classes	2	6	2	
Create security classes	2	8	3	
Create user classes	5	8	4	Yes
Create database tables in MySQL	0	6	5	
Design database tables	0	11	6	
Design user model	0	10	8	
Design security model	0	∞	10	
Design content model	8	10	11	Yes
Create data connector class	0	11	14	Yes

Table 5: Ten completed activities (Hit rate = 70%)

Task	Low Est. (hours)	Adj. Low	High Est. (hours)	Adj. High	Actual (hours)	Missed Prediction Interval?
Design FAQ model	0	0	4	5	2	
Create FAQ classes	2	1	6	7	2	
Create security classes	2	1	8	9	3	
Create user classes	5	4	8	9	4	
Create database tables in MySQL	0	0	6	7	5	
Design database tables	0	0	11	12	6	
Design user model	0	0	10	11	8	
Design security model	0	0	∞	∞	10	
Design content model	8	7	10	11	11	
Create data connector class	0	0	11	12	14	Yes

Table 6: Adjusting for width bias (Hit rate = 90%)

The next activity assigned to the software engineer is to “Design the user model.” Since only one item exists in the historical record, the first SISE step (sorting) is complete by default. Our software engineer hides all but the first column and compares the future activity’s description to the task description in the historical record. He decides that designing a user model is easier than designing a security model; we have a larger historical task, but no smaller one. The estimate, therefore, is a prediction interval of [0,10]. Our confidence in the estimate is equal to the hit rate, which is currently 100%.

Work proceeds and the activity is completed in eight hours. The estimate and actual are recorded and the new hit rate is calculated to be 100% (see Table 2). For convenience, the historical data in these examples will be kept sorted from smallest to largest task.

The third activity is assigned to the software engineer: “Design the content model.” Our software engineer scans the historical record, after hiding the numeric values, and decides that “designing a user model” is smaller and “designing a security model” is larger. Therefore, the prediction interval for the future activity is set at [8, 10]. The work is completed with an actual effort of 11 hours, giving a new hit rate of 67%, with two of the three completed tasks falling within his prediction intervals (see Table 3).

A fourth activity is assigned to the software engineer: “design database tables.” By scanning the historical record’s task descriptions, the software engineer decides the confidently larger task is “design content model,” but is unable to designate a smaller task. The prediction interval, therefore, is set as [0, 11].

The confidence level is assumed to be 67%, based on the historical hit rate. After referring to the sections on adjusting for bias, the software engineer considers making a shift adjustment. A one-hour upward shift of all the historical prediction intervals would move the hit rate from 67% to 100%. This leaves the estimator with two choices. The estimate’s prediction interval could be shifted one hour upward to account for a possible historical bias, or the estimate could be left alone. In short, the estimator now has two options to choose from: [0, 11] with a 67% confidence level or [1, 12] with a confidence level of 100%. Assume the estimator chooses to not shift the estimate due to the small data set size; the work is performed and recorded (see Table 4).

Assuming the software engineer proceeds in this fashion, he will accumulate a sizable historical record. With each hit or miss within the prediction interval, the hit rate will rise and fall. The software engineer may, at some point, choose to adjust a future estimate for width bias in order to increase his confidence level in a new estimate. Here’s a simple example, assuming ten completed tasks, with no verifiable shift bias to correct

As Table 5 indicates, the hit rate is 70%, with three of the ten tasks falling outside their prediction intervals. A future activity, "Create Contact Us page," has been assigned a prediction interval of [2, 8] and the confidence level is assumed to be 70%. In this case, however, the manager has requested a higher confidence level. To accomplish this, the software engineer adjusts for width bias.

The margins of error for each of the three tasks are one hour, one hour, and three hours, respectively. If the prediction intervals for all historical tasks were increased by one hour in each direction, the hit rate would rise to 90%. See Table 6.

Therefore, the prediction interval for the future activity "Create Contact Us page" must also be adjusted using a one-hour expansion, making it [1, 9] with a confidence level of 90%. In summary, the software engineer has a choice between two, fact-based estimates: [2,8] with a 70% confidence level or [1,9] with a 90% confidence level.

Each of the subsequent iterations through the SISE model follows a similar pattern to those reviewed above. The software engineer is assigned a new activity to complete. The activity is compared to previously completed tasks to identify a smaller and larger task, which leads to a prediction interval. The prediction interval is adjusted, if necessary and possible, to achieve a desired confidence level or prediction interval.

Validation of SISE

The SISE model has been validated through a multi-step process. First, over 100 software engineering students participated in a relative sizing activity, where they were asked to identify the larger of two tasks, based solely on the task descriptions. The results demonstrated that a majority of students were able to identify the larger task two-thirds of the time. Equally as important, the results indicated that students, on average, were unlikely to incorrectly identify a task's size; instead, they tended to identify the tasks as similar in size.

The next step in validating SISE involved sizing estimates using classroom programming assignments. Each student constructed a SISE-style estimate, as well as, an estimate based on a proxy-based model, derived from PSP's PROBE model. Overall, the SISE model's predictions proved no more or less accurate than the proxy-based approach. In addition, the students indicated that SISE, in their opinion, took less time and was based on less a complex model.

Additional validation of the SISE model within an industrial setting is planned for the near future. This will provide an opportunity to view the performance of SISE in a less structured (i.e. non-academic) environment over a longer term. In addition, an industrial environment will provide critical feedback on SISE's ability to integrate into a team environment, as the individual estimates are rolled up into team and project-level estimates.

Removing Shift

Shift bias involves a prediction interval that is too low or too high and may be corrected by shifting the interval. Shift bias exists only if the historical actuals fall predominantly below or above the associated prediction intervals; estimation error that is spread equally between overestimates and underestimates is a width bias and must be corrected in a different manner.

To determine if a shift bias exists, a form of simulation must be conducted. The simulation involves (1) compiling a list of the historical estimation error values, (2) shifting all the historical

prediction intervals by each error value, then (3) checking the change in overall hit rate with each shift.

Consider, for example, the following historical data in Table 7.

Table 7:

Activity	Prediction Interval (hours)	Actual (hours)	Error
Task 1	10-15	16	1
Task 2	12-16	18	2
Task 3	2-5	5	0
Task 4	1-3	3	0

The hit rate for the unmodified data set is 50%. All the prediction intervals could be shifted by 1 hour, which would cause Task 1 to become a successful prediction. Additionally, all the prediction intervals could be shifted by 2 hours, which would cause Task 2 to become successful. But how would these shifts affect the other predictions?

If all the prediction intervals are shifted by 1 hour, the hit rate rises to 75%; task 1's prediction interval now contains the actual effort and Tasks 3 and 4 are still successful. If the intervals are shifted by 2 hours, the hit rate rises to 100%. So, given this limited data set, shifting future estimate's prediction intervals by 2 hours may produce more accurate results.

Removing Width Bias

Once shift bias has been accounted for, the estimator may wish to either improve their precision or confidence level. This action involves a trade-off since increasing one reduces the other. For example, if the estimator wishes to increase their confidence level, the prediction intervals must be widened, making the estimates less precise. If the estimator wished to increase the precision of their estimates, by reducing the size of the prediction interval, the confidence level in the estimate will be proportionally reduced.

Improving the confidence level is accomplished by symmetrically widening all past prediction intervals by whatever amount is necessary to reach a hit rate equal to the desired confidence level. For example, if the historical record demonstrates a hit rate of 50% and the estimator would like to reach a confidence level of 80%, then all the past estimates' prediction intervals are widened until 80% of the actuals fall within the associates prediction intervals.

The inverse operation may be performed to improve the precision of the estimates. Past prediction intervals may be symmetrically reduced in size until the desired prediction interval width is reached. The new (and reduced) confidence level may then be calculated by checking the hit rate for the entire historical record.

Table 8 shows an example of how widening the prediction interval may allow for an increase in the hit rate from 60% to 80%.

Shifting the prediction intervals would not have improved the hit rate; however, if all the prediction intervals are increased by two hours (-1 to the low and +1 to the high), the hit rate moves from 60% to 80%.

Measuring Accuracy

The accuracy of a single value estimate is determined by the magnitude of the estimate's error, relative to the actual effort. For example, if an activity is estimated to take 4 hours, but actually takes 5, the magnitude of relative error (MRE) is 0.2 (or 20%). Here is the formula:

Table 8:

Activity	Original Prediction Interval (hours)	Actual (hours)	New Prediction Interval (hours)
Task 1	10-15	10	9-16
Task 2	12-16	16	11-17
Task 3	5-7	6	4-8
Task 4	9-11	12	8-12
Task 5	13-15	11	12-16

$$\text{MRE} = (\text{actual-estimate})/\text{actual}$$

When using prediction intervals to describe an effort estimate, the practitioner's accuracy is determined by the number of activities with actual effort values that fall within the predicted interval. Here's the formula:

$$\text{Hit Rate} = \text{No. hits} / \text{No. estimates}$$

For example, consider the list of work activities in Table 9.

Eight of the ten activities were completed within the time frame defined by the prediction interval; Tasks 4 and 5 took more and less time, respectively, than predicted. Therefore, the hit rate for this sample is 0.8, or 80%.

Table 9:

Activity	Original Prediction Interval (hours)	Actual (hours)
Task 1	10-15	12
Task 2	12-16	15
Task 3	2-5	5
Task 4	1-2	3
Task 5	16-22	15
Task 6	9-13	12
Task 7	4-6	5
Task 8	6-8	8
Task 9	3-4	4
Task 10	6-10	9

Conclusion

The SISE model represents an empirically based approach to effort estimation that relies less on complex mathematical models and more on intuitive expert judgment, without sacrificing the quality of the final product. Software engineers willing to take the first tentative steps toward adopting a personal process now have access to a truly lightweight, agile estimation model. The SISE model does not burden the practitioner with any more work than the absolute minimum necessary to produce a reasonably accurate, fact-based effort estimate. In addition, the model is the first of its kind, suitable for use by a single software engineer.

Further development and improvements to the model are currently underway at Auburn University's microISV Research Lab. We are formalizing ways in which the SISE model may be integrated into team-based software processes, as well as tool development. ♦

NOTES

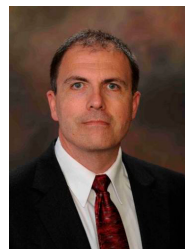
1. M Jorgensen, K H Teigen, and K J Molokken-Ostfold. Better sure than safe? Over confidence in judgment based software development effort prediction intervals. *Journal of Systems and Software*, 70(1-2):79-93, 2004.
2. Russell Thackston and David Umphress. Individual effort estimating: Not just for teams anymore. *CrossTalk: The Journal of Defense Software Engineering*, 25(3):4-7, May/June 2012.
3. Pomeroy-Huff, Marsha, et al. "The Personal Software ProcessSM (PSPSM) Body of Knowledge, Version 2.0." Software Engineering Institute. Carnegie Mellon, August 2009. Web. 30 October 2013.
4. Russell Thackston and David Umphress. Micropreneurs: The rise of the microISV. *IT Professional*, 15(2):50-56, 2013.

ABOUT THE AUTHORS



Russell Thackston is an Assistant Professor of IT at Georgia Southern University. He earned his Ph.D. in computer science and software engineering from Auburn University. His research interests focus on software process and effort estimation, specifically with regard to individual software engineers and microISVs. His work on SISE -- an effort estimation model tailored to individuals -- has been published in *Crosstalk*, the *Journal of Defense Software Engineering*, and his research into microISVs has been published in *IEEE's IT Pro*. Russell served in the U.S. Air Force during Desert Storm and Desert Calm and has been happily married for more than 25 years.

Phone: 912-478-4218
E-mail: rthackston@georgiasouthern.edu



David A. Umphress, Ph.D., is an associate professor of computer science and software engineering at Auburn University, where he specializes in software development processes. He has worked over the past 30 years in various software and system engineering capacities in military, industry, and academia settings. Umphress is an Institute of Electrical and Electronics Engineers (IEEE) certified software development professional.

Phone: 334-844-6335
E-mail: david.umphress@auburn.edu

The Effectiveness Formula¹

Key to Productivity Improvement

Dr. Randall W. Jensen, Consultant

Abstract. The Effectiveness Formula is based on three development attributes of success: communication, management effectiveness, and technology. The formula in its generic form can be applied to any organization in any industry.

"Our job is to escape the cave, look around, then come back and tell others what we have seen...Of course, they won't believe us." Daniel K. McKiernan

Introduction

The logical place to begin a discussion about software engineering is the "software crisis" at the 1968 NATO conference in Munich, Germany. A list of software problems was presented as the major development concerns at the NATO Conference. The problem list included software that was:

- Unreliable
- Delivered late
- Prohibitive in terms of modification costs
- Impossible to maintain
- Performed at an inadequate level
- Exceeded budget costs

This list of problems still persists in much of the software development industry some 40 years later.

The term "software engineering" was first used in 1968 as a title for the world's first conference on software engineering, sponsored and facilitated by NATO². The conference was attended by international experts on software who agreed that the discipline of software engineering was needed to address poor quality of software, get projects exceeding time and budget under control, and ensure that software is built systematically, rigorously, measurably, on time, on budget, and within specification. One of the significant conference outputs was a software engineering college curriculum that just happened to be identical to the standard computer science curriculum.

Software engineering is defined as (1) the systematic design and development of software products and (2) the management of the software process. Software engineering has as its primary objectives the production of programs that meet specifications, and are demonstrably accurate, produced on time, and within budget.³

All organizations can be represented by the people-process-project triad shown in Figure 1. The projects, which vary from industry to industry, are represented by the Project node. All organizations contain a People node representing the physical environments the people are part of, and the management environments of those individuals. The Process node of the triad

represents the development processes used to produce the organization products and the management of those processes. Organization capability and development productivity are largely driven by the communication and management attributes of the People node of the triad model.

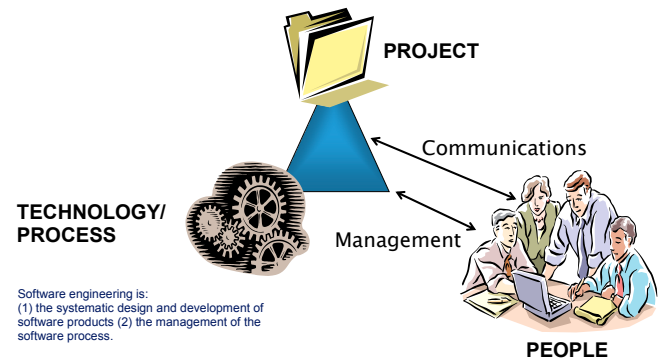


Figure 1 Software development triad: Projects, Process, People

For the purpose of this presentation I am going to divide the processes into two camps. The first camp includes "traditional" processes represented by the classic waterfall and spiral⁴ development and other variations of the waterfall process. The second camp is comprised of the varied agile software development processes. Agile methodologies include Pair Programming⁵ (1975), Scrum⁶ (1995), Crystal Clear⁷ (1996), Extreme Programming⁸ (1996), among others. A major agile characteristic is stated in the Agile Manifesto⁹ as "We value individuals and interactions over processes and tools." The traditional camp can be described as those who value processes and tools over individuals and interactions.

The people node is common to both traditional and iterative Agile process camps. For example, pair programming, is very dependent on communications and management support to function, and the pair programming concept works well in a traditional environment. Barry Boehm wrote in 1981 that: "Poor management can increase software costs more rapidly than any other factor..."¹⁰

Weinberg's Second Law of Consulting¹¹ added a supporting observation: "No matter how it looks at first, it's always a people problem."

Looking deeper into the list of software problems, we find that the perceived solution to the software development problem is software engineering, or technology. According to the results from the 2013 Standish Chaos Manifesto¹², technology has not been the total solution to project success. The Chaos report divides projects into three classes: successful, challenged, and failed. About 39 percent of the 2012 projects evaluated were successful. Forty three percent were delivered, but with significant average overruns of near 59 percent of cost and 74 percent of schedule while delivering only 69 percent of the original requirements (challenged). Still about 18 percent were cancelled before delivery (failed).

The Effectiveness Formula

Thirty-five years ago, Chuck Tonies and I wrote about the state of software engineering as a basis for what we called the "Effectiveness Formula." As I reviewed the introduction to our 1979 book *Software Engineering* for this presentation, I was

struck by the similarity between our description of software engineering in 1975 and the state of the profession today. The lessons we learned in our software development work are directly applicable to both engineering and development in any industry.

Most development activities, including software system development, are dynamic. No matter how effective the development methods and the process, and no matter how stable the project staff, some degree of rethinking, replanning, redefining, and redirection are necessary as a project proceeds. Unfortunately, communication among team members is not always perfect. An incomplete or incorrect understanding of requirements, designs, and interfaces is inevitable. Frequent communication among all participants on a development project is the only way to prevent or correct such misunderstandings.

If people are not capable of participating (or are not motivated to participate) in the inevitable ebb and flow of management decisions on a project, or if they can't communicate daily with members of the team, the value of their contributions (no matter how brilliant) is diminished, because those contributions probably don't match the real product requirements.

We developed a conceptual model of the software development organization and its embedded interactions to illustrate the software development process. The simple model highlights the interactions that are always present in any development organization. A person's value to an organization operating in the industrial environment is dependent on three attributes: (1) available technology, (2) the ability to understand management concepts, and (3) the ability to communicate. The model shows these qualities are so intimately involved in the development process that the net effect of an individual's effort is best represented by the product of the Effectiveness Formula attribute values:

$$E = C[M(T)] \quad (1)$$

where

E = Net effectiveness (0–1)

C = Communication ability and skills (0–1)

M = Management concept awareness (0–1)

T = Technical ability (0–1)

Our experience in the software industry, and especially in product-oriented environments, has shown the formula to be a realistic model of software engineering performance. This effectiveness (performance) model can be applied equally well to any industrial development endeavor. If any of the attributes have a value of zero, the net effectiveness of the organization is also zero. If all of the attributes are 0.5, the effectiveness of the organization is only 0.125. Software engineering, by definition, does not include either management or communication.

We are still in an age of technical specialization; however, software development work is by its very nature a complex interactive process that involves much more than technology. It requires careful, intense management, and even the most specialized of the contributors must act in concert with his colleagues and the management plan if the development process is to be efficient.

The software development industry, especially within the U.S. Department of Defense (DoD), also collects enormous amounts of data to improve development estimates, and that

stockpile provided much of the data behind the refinement of the Effectiveness Formula today. The organization-related parameters of both the Seer and COCOMO software cost estimating models can be mapped directly into this conceptual model.

I have been collecting software cost-analysis data from across the Department of Defense community since 1972, to calibrate cost- and schedule-estimating models. Within these records, dating back to the 1960s, I noted a constancy of the capability data for each of the DoD contractors, changing very little over the 50-year period continuing into the 2000s.

As an example of the use of the Effectiveness Formula as a

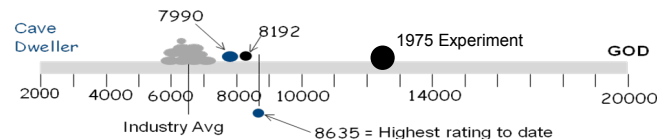


Figure 2 Capability Measures for traditional and agile environments

capability measure, the data in Figure 2 represents the distribution of Seer-based developer basic technology constant¹³ values across the range of 2000 to 20000. The clustering of the project data around the 6500 value shows the measured consistency of the organization capabilities across the industry. The data cluster at 6500 is also related to traditional developments with low communication and management model values. The data points between 7990 and 8635 represented projects with higher management ratings. The 1975 data point represents the first measured pair programming project.

The first rather obvious observation in the data is that most software organizations are using similar or identical development technologies. The organizations adopted similar methods and tools at roughly the same time. Each technology improvement offered some capability improvement as shown in Figure 2, even though the capability gains are not strikingly significant.

The second observation is that the management approaches across the clustered organizations are almost identical, all being slight variations of classical management best described as Theory X¹⁴. The five management functions (planning, organizing, commanding, coordinating and controlling) introduced in 1916 by Fayol¹⁵ are still the focus of management training today. The development environment for these organizations all provided limited communications.

Table 1 shows the relationship between the Effectiveness Formula values from the completed project data used to generate Figure 3 to the relative organization capability ratings (percentile) of the software development organization database.

Basic Technology Constant (C_{bt})	Effectiveness Value	Relative Percentage
2000	0.11	10
6500	0.25	50
8192	0.38	70
8635	0.40	78
12500	0.5	88

Table 1 Comparison of Basic Technology Constant (C_{bt}), Effectiveness value and relative ranking of project data

The basic technology constant data in the table can be related directly to the Effectiveness Formula values.

The traditional software development organization capability ratings are remarkably clustered around an Effectiveness Formula value of approximately 0.25, or about 6500 on the familiar basic technology constant Ctb value. The relative percentile ranking of such an organization is the industry average (50%). Assuming a traditional organization utilizes a fairly advanced technology attribute value of 0.8, the combined impact of communication and management of that organization must be a low combined value of approximately 0.3 to achieve an effectiveness value of 0.25.

The Attributes

Success in software development productivity improvement is not a matter of concentrating one's effort in any one of the three attributes that comprise the Effectiveness Formula. A narrow view of the importance of any one of the attributes spells failure for an organization attempting to make improvements.

Technology

The first part of the software engineering definition consists of the set of methods and tools we use to develop the ultimate product. Technology has existed since the time of the agrarian plow, and it is constantly evolving to support the needs of the developer. Better tools and methods contribute to improving effectiveness and efficiency as one of the three primary attributes of the Effectiveness Formula.

There have been several development technology breakthroughs during the past forty years that have significantly decreased the cost of software products. For example, the introduction of FORTRAN and COBOL decreased the cost of a given product functionality to one-third of the cost achievable when implemented in Assembler due to the decrease in the source lines of code to required achieve the product functionality. The transitions from C++ to the newer visual languages, and the advent of object-oriented structures, created significant software cost savings because the required number of source lines have continually decreased. However, when we look at the effort required to produce a single line of source code in any given programming language, we see that traditional software development productivity (measured from start of development through delivery or software-system integration) has increased, with little blips and dips, almost linearly at the rate of less than two source lines per person-month (lppm) per year as shown in Figure 3.

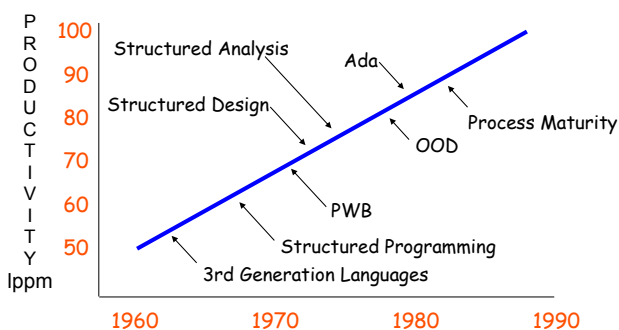


Figure 3 Traditional software development productivity gains -- 1960 to 1990

We have been learning new things about software development during this period. The development environment focus was almost entirely on the product during the 1960s and early 1970s. The principle activity, once the requirements were established was programming. Software development technology, namely programming languages, improved as the system requirements grew to manage the size and complexity of the tasks increased. Development platforms improved to support the ever increasing size of software systems.

The second part of the software engineering definition formalizes the development process. Technology was the primary focus in the 1960s, when the software development discipline was new. The early 1970s brought a shift in focus to the development process. In the mid-1980s, the Software Engineering Institute introduced the CMM® as an approach to stabilize the development process, improving quality and productivity by focusing energy on process improvement. The CMM concept was introduced in 1987 by Watts Humphrey¹⁶. Humphrey stated: "While there are many unique characteristics to software, they all require more management discipline, not less. Managers should thus demand detailed plans, tracking systems, and periodic technical and management reviews. Software management should be entirely traditional, only more so. Unfortunately, many managers who insist on these items for hardware let their software teams get by without them."

Humphrey's remark clearly conflicts with the agile focus on individuals and interactions over processes and tools. The CMMI® Guidelines for Process Integration and Product Improvement place a rigid outline for managing the development of software products without filling in the details of how the process is implemented. Process provides no support for the second and third attributes of the Effectiveness Formula; that is communications and management.

Management of the process, or Process maturity, is not a programming language, not a development method or approach, nor is it a tool set that supports the development, but it is a major factor in any engineering approach. CMMI¹⁷ (Capability Maturity Model Integration) is a process improvement model for the development of products and services consisting of best practices that address development and maintenance activities covering the product lifecycle from conception through delivery and maintenance.

Communication

Broadly defined, communication is the act or process of exchanging information between individuals, using a common system of symbols, signs, or behavior. The related definition of collaboration is working jointly with others, especially in an intellectual endeavor. Both elements, communication and collaboration, are necessary to produce a software product effectively and efficiently.

The most important observation is the norm for physical environments in current traditional development. The most common modern large-scale software development environment is the "cube farm," in which all communication between developers is carried through the organization's computer network. The cubicle development environment is not designed, by intent or by chance, to foster interactive communication among the project participants. Instead, the cube farm's purpose appears to even the casual observer as a means to prevent communication. The

lack of effective communication blocks any attempt to motivate the staff, prevents collaboration within the staff, and eliminates the potential for forming teams.

The least common environment, the “skunk works,” is typically defined as a small group of experts who move outside an organization’s mainstream operations in order to develop a new technology or application as quickly and efficiently as possible, without the burden of the organization’s bureaucracy or strict process application. The skunk works workspace is a physically open environment that encourages intra-team access and communications. Tools and processes are tailored and adapted to the project’s requirements. An Agile environment fits into this type of structure.

The effectiveness of voice or visual communications is supported by a well-known research study by Mehrabian and Ferris¹⁸ shown in Figure 4. The information transfer effectiveness, however, is diminished when we remove any source of information. For example, we can remove the visual part of the information transfer by forcing the communicators to use a telephone. This eliminates all of the gestures, body language, and eye contact from the conversation and lowers the information transfer by over 50 percent. This important information source is no longer available to reinforce understanding between the two individuals and leads to gaps in communication, as well as misunderstandings.

Removing the visual and vocal information elements of the discussions leaves us with only 7 percent of the information content. Information transfer is significantly degraded when we rely solely on paper because we remove the ability to ask and respond to clarifying questions. We lose not only the subtle elements of voice communication, but also the real-time elements necessary for feedback between one another.

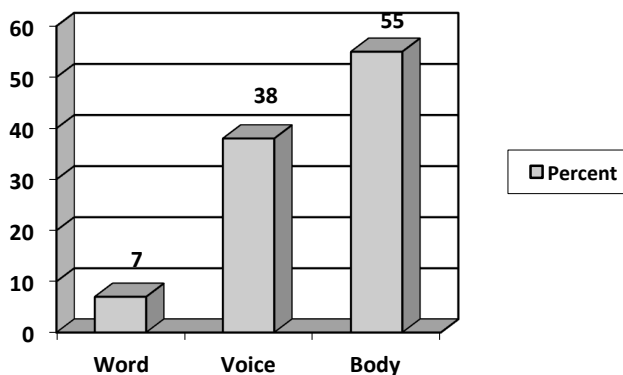


Figure 4 Components of communication.

By placing each software engineer contributing to a product development in a 36 to 64 square foot cubicle lowers the communication attribute of the net Effectiveness Formula to less than 0.07. Some argue that by connecting the software engineer’s boxes by a high-speed network connection restores the information content to a level approaching 100 percent in spite of removing the visual and vocal communication elements. The argument is unsupportable. To further degrade the effective communication, consider removing the real-time feedback element as well. Communication between workstations is little more effective than communication by written documents.

The solution to communication barriers is not modern technology, such as the use of e-mail and high-speed network commu-

nications. These solutions are often proposed for local communication support and to justify remote software development teams. Ironically, this technology solution raises even greater barriers than the cubicle example. At least people in adjacent cubicles have some physical contact. Remote locations are sometimes separated by thousands of miles.

Communication, which has a range of values between zero and (ideally) unity, has an effective value of only 0.07 in a cube farm. With a value that low, even a management effectiveness value of 1.0 and a perfect technology value (1.0) cannot provide much of a contribution to the person’s effectiveness.

Management

Project management theories have been around since long before the beginning of the 20th century. In the early 1900s, the focus of management studies centered on the most effective methods to organize and structure the industrial organization; that is, ways to organize, delegate, and coordinate work efficiently. Five basic functions of management were identified: planning, organizing, commanding, coordinating, and controlling. These five functions, which were defined in 1911, appear in almost all the current management literature.

The well-known Hawthorne experiment conducted in the 1920s showed that the solution to the productivity dilemma was not found in the physical working conditions, but in the human aspects. The most significant factor affecting productivity in an organization was found to be interpersonal relationships developed on the job, not just pay and working conditions. This factor is widely referred to as the Hawthorne effect. When informal groups identified with management, productivity rose. The increased productivity reflected the workers’ feeling of competence—a sense of mastery over the job and the environment. The study also showed that when the workers’ goals were in opposition to those of management as often happens with micro-management, productivity remained at only marginally acceptable levels, or decreased from the norm.

The Hawthorne experiment was a forerunner of the development of Douglas McGregor’s classic Theory X-Theory Y view of management. McGregor proposed that there are two primary categories of organizational management thinking¹⁹, each with pronounced impacts on the way organizations function. Theory X assumes that most people prefer to be directed, are not interested in assuming responsibility, and want safety (job security) above all. Theory X corresponds to the belief that most people are motivated by money, fringe benefits, and the threat of punishment.

Managers who follow Theory X assumptions attempt to structure, control, and closely supervise their workers. These managers believe that external control is clearly appropriate for dealing with unreliable, irresponsible, and immature people. The mode is consistent with the five advertised functions of management.

McGregor’s alternate theory of basic human behavior, Theory Y, assumed that people are not lazy and unreliable by nature. They can be self-directed and creative if properly motivated. McGregor concluded that it is management’s responsibility to free the potential of the workers so that they can achieve their own goals. Supportive Theory Y managers provide the means to achieve organizational goals, as opposed to Theory X managers, who control and closely supervise workers. I find the X-Y comparison to shepherders and shepherds.

In spite of the work by these behavioral pioneers and many others, software management remains primarily a Theory X culture. I am frequently reminded of Weinberg's Second Law of Consulting²⁰: No matter how it looks at first, it's always a people problem.

Summary

Software engineering is, by definition, contained in the Process node of the triad depicted in Figure 1. It is also totally contained in the technology attribute of the Effectiveness Formula. Looking back as far as the 1968 conference on software engineering, it has offered little or no support for capability and productivity improvement. If improved productivity is our goal, software or otherwise, effective leadership should be our focus. The Effectiveness Formula is based on three development attributes of success: communication, management effectiveness, and technology. The formula in its generic form can be applied to any organization in any industry. Fortunately, the software industry has collected sufficient data to apply a quantitative measure to organization productivity and capability.

One important feature of management that is not discussed by McGregor and others is that people must have contact with each other in order to achieve associated social, esteem, and self-actualization goals. Isolating people, such as is accomplished with cubicles, negates the positive aspects of Theory Y. Leadership attitude toward employees allows the organization's communication to grow, thus increasing productivity. You can only imagine a manager trying to lead a group of engineers, none of whom know how their work fits into the system concept or direction being followed by the other engineers. Without communication between employees, productivity improvement is difficult or even impossible to achieve.

A final point to be made in the Effectiveness Formula is the quantitative support of the measure. The conceptual model attribute ranges were simply 0 to 1. A quantitative implementation of the model shown in Table 1 underlies both the proven COCOMO and Seer (SEER-SEM) software cost estimating models.

Disclaimer:

CMM® and CMMI® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. ♦

ABOUT THE AUTHOR



Randall W. Jensen, Ph.D., is an independent software acquisition consultant with more than 50 years of practical experience as a computer professional in hardware and software development. For the past 40 years, he has actively engaged in software engineering methods, tools, quality software management methods, software schedule and cost estimation, and management metrics. He retired as chief scientist of the Software Engineering Division of Hughes Aircraft Company's Ground Systems Group (1993) and a subject matter expert from the USAF Software Technology Support Center (2011). He developed the model that underlies the Sage and the Galorath, Inc.'s SEER-SEM software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering" (1979), "Improving Software Development Productivity: Effective Leadership and Quantitative Methods in Software Management" (2014), and numerous software and hardware analysis papers. He has a BS, an MS, and a Ph.D. in electrical engineering from Utah State University.

E-mail: costsage@comcast.net

NOTES

1. Jensen, R. W., *Improving Software Development Productivity: Effective Leadership and Quantitative Management*, (Englewood Cliffs, NJ: Prentice Hall, 2014)
2. Naur, P., and Brian Randell, *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmish, Germany, October 7-11, 1968*
3. Mills, H.D., "The management of software engineering, Part I: Principles of software engineering," *IBM Systems Journal*, vol.19, no.4, pp.414,420, 1980
4. Boehm, B. W., "A Spiral Model for Software Development and Enhancement, *IEEE Computer*, May 1988, pp. 61-72
5. Jensen, R. W., "A Pair Programming Experience," *CrossTalk*, March 2003)
6. Schwaber, K., and M. Beedle, *Scrum: Agile Software Development*, (Upper Saddle River NJ: Prentice-Hall), 2002.
7. Cockburn, A., *Crystal Clear: A Human-Powered Methodology for Small Teams*, (Reading, MA: Addison Wesley), 2004
8. Beck, K., *Extreme Programming Explained*, (Boston MA: Addison-Wesley), 2001
9. Fowler, M. and J. Highsmith, "The Agile Manifesto," *Software Development*, August, 2001
10. Boehm, B. W., *Software Engineering Economics*, (Englewood Cliffs, NJ: Prentice-Hall, Inc.) 1981, pp. 486
11. Weinberg, G. W., *The Secrets of Consulting*, (New York, NY: Dorset House Publishing), 1985, p. 5.
12. Standish Group International, *The Chaos Manifesto*, (West Yarmouth, MA: Standish Group International), 2013
13. Jensen, R. W., "A Macrolevel Software Development Cost Estimation Methodology," *Proceedings of the Fourteenth Asilomar Conference on Circuits, Systems, and Computers*, Pacific Grove, CA, November 17-19, 1980
14. Hersey, P. and K. Blanchard, *Management of Organizational Behavior*, (Englewood Cliffs, NJ: Prentice-Hall, Inc.), 1977, P.55
15. Fayol, H., *General and Industrial Management* (London: Pitman, 1916)
16. Watts S. Humphrey, *Managing the Software Process*, (Addison-Wesley, Boston, MA), 1989
17. Chrissis, M. B., M. Konrad and S. Shrum, *CMMI: Guidelines for Process Integration and Product Improvement*, 2nd Ed., (Upper Saddle River, NJ: Addison-Wesley), 2007
18. Mehdrabian, A. and S. R. Ferris, "Inference of Attitudes from Nonverbal Communication in Two Channels," *Journal of Counseling Psychology*, Vol. 31, 1967
19. Hersey, P. and K. Blanchard, *Management of Organizational Behavior*, (Englewood Cliffs, NJ: Prentice-Hall, Inc.), 1977, P.55
20. Weinberg, G. M., *The Secrets of Consulting: A Guide to Giving & Getting Advice Successfully*. New York, NY: Dorset House, 1986, p. 5.

Meeting Cancelled!

The old proverb, as originally recorded in Benjamin Franklin's Farmer's Almanac, states that "a rotten apple spoils the barrel", particularly true for meetings in the sense that a few long unproductive conferences tarnishes the popular image of this otherwise effective managerial tool. Experience has shown that meetings run the whole range of effectiveness—from the very informative to the least helpful with majority of meetings falling somewhere in between. Clearly, most meetings are a good investment of time and they allow employees to better do their jobs more effectively. Unfortunately, the few meetings, which waste our valuable time and sap our enthusiasm, seem to catch our collective attention to the point that we delight when they are cancelled. For the purposes of this article, I would ask the reader to wipe the slate clean and review meetings with the aim of improving them. Certainly the incentive is there to make meetings productive since they consume so much of the employee's time (Fortune magazine estimates that personnel spend 15% to 25% of their time in meetings, which makes it incumbent upon organizations to make certain that their meetings are efficiently run and are value-added). Thus, it is obvious that meetings are here to stay, but there is much that we can do to make the most of our time spent in attending meetings; That is, we should go to meetings and listen attentively when you are ordered by your boss, work hard to improve meetings when you are the organizer, and strive to use online (distance) conferencing whenever possible to save valuable travel time and expense.

Let us start with the most important reason that we attend meetings—because our boss directs our participation. While we usually have little control of these leadership-originated meetings, it is nevertheless important to listen carefully and note down critical information. It is a good investment of your time to learn firsthand of governing policies and the context these decisions are being made by your employer. These meetings provide a good way to forecast the future health of your organization and to determine for yourself on how to protect your own career.

Next, let us turn to meetings that you may control or have at least partial influence on how these conferences are set up and run. Here are some helpful tips on how to improve your meetings based on personal experience as well as the collective wisdom found in management literature:

1. Maintain Meeting Control—Start with a reasonable agenda to set boundaries for the meeting and work hard to stay on track by keeping off-the-topic conversations to a minimum. Do not be afraid to gently challenge people who monopolize the air waves or go off to very narrow areas of interest.

2. Set Reasonable Meeting Duration—My experience has been that most meetings can be concluded within 30-45 minutes (60 minutes being the upper limit for 90% of meetings).

3. Invite the Right People—It is critical to invite people with the domain expertise and authority to act to a meeting while leaving out people with tangential interest in the meeting topic.

4. Get Feedback from Meeting Participants—Ask for private feedback from a select set of trusted gray beards (or experts) on how the meeting progressed. Let us face it—you may be screwing up in spite of your best efforts, but would not know it unless someone you trust points it out to you.

5. Provide Meeting Summaries and Action Item List—Preparing these documents will help summarize the meeting for those who did not attend and set the stage for follow-on action. Circulate these documents to all meeting attendees. Remember that these documents are a good investment of your time and will increase chances of your meeting's success.

These suggestions are not a panacea or a cure-all to make meetings perfect each time, but these are common sense ways to make meetings more productive.

While this article focuses primarily on face-to-face meetings, the need to make certain that online or distance conferences are productive is equally required. Consider that almost every face-to-face meeting taking place today includes a telephone or internet link to allow distant attendees to participate in the proceedings. The need for firm start and finish times, clear agenda established before the meeting, and a moderator to keep the meeting on track becomes even more pronounced because distant attendees usually lack the ability to see visual cues and gauge the general mood of the meeting attendees that in-person attendees use to judge when and how much to speak. In my many years with teleconferencing, I have found that it is too easy for participants to start pontificating about some point that is "critical" only from their own point of view or to start discussing a narrow area of concern that generally should be relegated to another venue. Hence, the need for meeting control becomes even more critical.

I want to conclude this article with my observation that meetings are not inherently good or bad, but depend largely on how they are organized. Based on my attendance of literally hundreds of meetings, I can tell you that meetings run the whole range of quality, but most can be improved with implementing some of the steps cited above. But remember that even the best advice will go unheeded unless you win over your leadership to the cause of making meetings more productive. This is the key to bringing positive change because the most frequent reason people do anything is because their management finds it to be important. Conversely, consider that deeply imbedded institutional practices are difficult to modify since it requires conscientious effort to replace less productive practices with more effective ones. Most of all, please keep in mind that even a modest improvement will make your meetings more valuable and everyone will win in terms of more efficient use of their time. Hopefully the resulting culture of holding efficient meetings will cause meeting attendees to dread that their favorite meeting has been canceled! ♦

Akbar Khan

ABOUT THE AUTHOR



Mr. Akbar Khan is a Software Systems Engineer with twenty-five years of experience and is licensed as a Professional Engineer in the states of New York, New Jersey, Maryland, and Delaware. He received his Bachelors of Science in Industrial/Systems Engineering from Polytechnic Institute of New York University and a Masters of Engineering in Systems Engineering from Stevens Institute of Technology. Currently, Mr. Khan is pursuing his Doctor of Science in Information Technology at Towson University. He is a native of New York City and is currently living near Baltimore, Maryland with his wife and three children.

Phone: 443-395-2191

E-mail: akbar.r.khan.civ@mail.mil

REFERENCES

1. How to Run an Effective Meeting. (n.d.). CBSNews. Retrieved April 4, 2014, from <<http://www.cbsnews.com/news/how-to-run-an-effective-meeting/>>
2. Lipman, V. (2013, March 1). 5 Simple Steps To More Efficient, Effective Meetings. Forbes. Retrieved April 3, 2014, from <<http://www.forbes.com/sites/victorlipman/2013/03/01/5-simple-steps-to-more-efficient-effective-meetings/>>
3. Nazar, J. (2014, April 3). Your Meetings Are Awful -- Here's How To Fix Them. Forbes. Retrieved April 4, 2014, from <<http://www.forbes.com/sites/jasonnazar/2014/04/03/your-meetings-are-awful-heres-how-to-fix-them/>>
4. "Rotten Apple." <Dictionary.com>. Dictionary.com, n.d. Web. 08 May 2014.
5. Section 1. Conducting Effective Meetings. (n.d.). Main Section. Retrieved April 4, 2014, from <<http://ctb.ku.edu/en/table-of-contents/leadership/group-facilitation/main>>

WANTED

Electrical Engineers and Computer Scientists *Be on the Cutting Edge of Software Development*

The Software Maintenance Group at Hill Air Force Base is recruiting **civilians** (*U.S. Citizenship Required*). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance, and time paid for fitness activities. **Become part of the best and brightest!**

Hill Air Force Base is located close to the Wasatch and Uinta mountains with many recreational opportunities available.



facebook

www.facebook.com/309SoftwareMaintenanceGroup

Send resumes to:
309SMXG.SODO@us.af.mil
or call (801) 777-9828



Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

11th ACM/IEEE- Symposium on Architectures for Networking and Communications Systems (ANCS'15)

7-8 May 2015
Oakland, CA
<http://www.ancsconf.org>

The 37th International Conference on Software Engineering

May 16-24, 2015
Firenze, Italy
<http://2015.icse-conferences.org>

Agile Development Conference West

June 7-12, 2015
Las Vegas, NV
<http://adcwest.techwell.com>

STARCANADA (Software Testing Analysis & Review Canada)

June 21-25, 2015
Vancouver, BC Canada
Westin Bayshore
<http://starcanada.techwell.com>

20th International Conference on Reliable Software Technologies

Ada-Europe 2015
22-26 June 2015
Madrid, Spain
<http://www.dit.upm.es/~ae2015>

CGI'15 32nd Annual Conference

24-26 June 2015
Strasbourg, France
<http://cgi2015.unistra.fr>

The 39th Annual International Computer, Software & Applications Conference

July 1-5, 2015
Taichung, Taiwan
<http://www.computer.org/portal/web/COMPSA>

The 27th International Conference on Software Engineering and Knowledge Engineering

July 6- July 8, 2015
Pittsburgh, Pa
<http://www.ksi.edu/seke/seke15.html>

Thirtieth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)

July 6-10, 2015
Kyoto, Japan
<http://lics.rwth-aachen.de/lics15>

29th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy

July 13-15, 2015
Fairfax, VA
<http://dbsec2015.di.unimi.it>

INCOSE 25th Annual Symposium IS 2015

13-16 July 2015
Seattle, WA
<http://www.incose.org/newsevents/events/details.aspx?id=255>

What exactly is Software Engineering?

Once upon a time, in a kingdom far, far away, a king summoned three of his subjects. One was a coder, the other a computer scientist, and the third a software engineer. The king showed all three a shiny metal box with two slots in the top. "Behold", said the king. "This is my toaster. The first slice of bread is too light, the second just perfect, but the third is too dark. I want you to make my toast consistently cooked!"

The coder replied that he could easily fix it with a simple microcontroller. He said "Using a four-bit microcontroller, I would write a simple program that reads the darkness knob and quantizes its position to one of 16 shades of darkness, from snow white to coal black. The program would use that darkness level as the index to a 16-element table of initial timer values. Then it would turn on the heating elements and start the timer with the initial value selected from the table. At the end of the time delay, it would turn off the heat and pop up the toast. Come back next week, and I'll show you a working prototype."

The second advisor, the computer scientist, immediately recognized the danger of such shortsighted thinking. He said, "Toasters don't just turn bread into toast, they are also used to warm frozen waffles. What you see before you is really a breakfast food cooker. As the subjects of your kingdom become more sophisticated, they will demand more capabilities. They will need a breakfast food cooker that can also cook sausage, fry bacon, and make scrambled eggs. A toaster that only makes toast will soon be obsolete. If we don't look to the future, we will have to completely redesign the toaster in just a few years. With this in mind, we can formulate a more intelligent solution to the problem. First, create a class of breakfast foods. Specialize this class into subclasses: cereal grains, pork, and poultry by-products. The classification process should be repeated with grains divided into toast, muffins, pancakes, pastries, and waffles; pork divided into sausage, links and bacon; and poultry divided into scrambled eggs, hard-boiled eggs, poached eggs, fried eggs, and various omelet classes. The ham and cheese omelet class is worth special attention because it must inherit characteristics from the pork, dairy and poultry classes. Thus, we see that the problem cannot be properly solved without multiple inheritance. At run time, the program must create the proper object and send a message to the object that says, 'Cook yourself'. The object itself will know if it consists of toast, eggs, pastry, etc.

Reviewing the process so far, we see that the analysis phase has revealed that the primary requirement is to cook any kind of breakfast food. In the design phase, we have discovered some derived requirements. Specifically, we need an object-oriented language with multiple inheritance. I suggest C++, or maybe Java if we plan on re-hosting to newer hardware in the future. Of course, users don't want eggs to get cold while the bacon is frying, so concurrent processing is required, too. Maybe Ada 2012 is the way to go. And – of course – the unit will need an intelligent camera and a sensitive weight unit, so we can determine the composition, weight, and derived density of bread and other breakfast objects. This is going to require a lot of processing – I'm thinking a i7, maybe a 128 Gig Solid State Drive, 4 Gig RAM, and a rock-solid UNIX-based OS.

We must not forget the user interface. Users won't buy the product unless it has a user-friendly, graphical interface. We need a good OS – perhaps UNIX. Once the system boots – the system could display a pull-down menu. Users can pull down a menu and click on the foods they want to cook. Specific tailoring options can then be selected (raisin bread, cinnamon bread, etc.)"

The king asked the software engineer his opinion. The software engineer suggested that the king banish the coder and analyst from the kingdom, and he taught the king how to adjust the toaster – turning it up a bit for the first slice, down a bit for the second slice, and down even further for subsequent slices. And they all lived happily ever after. THE END.

What is software engineering? Well, I have taught "software engineering" at the college level for 29 years now. I would imagine, if you have made it to this point in this issue – you probably have an idea of what software engineering is for yourself. In my college classes, we teach such diverse topics as risk analysis, cost and time estimation, quality assurance, and configuration management. I also teach lifecycles, coding techniques and approaches (functional, object-oriented, etc.) I teach good documentation (lifecycle and code), design techniques, and how to perform testing. This is a required class; all Computer Science majors take my class.

I have the luxury of teaching a second class ("Requirements Analysis and Design") to the majority of my CS majors (not required – but over 90% decide to take the second class). In this class, we perform more in-depth analysis and design, and talk more about the non-coding part of developing software.

So, when they graduate – are my students software engineers? Nope – not even close. Because any project that I can comfortably cover in a one-semester course is way too small to give them a real taste of what the real world is like.

I heard from a lot of my former students after they have been "in the profession" for several years – and they always say "I thought you were kidding with your stories. Actually, it's worse in the real world than we could have imagined." However, the basic skill set I teach them seems to give them the tools and knowledge they need to keep them employed until they gain the experience they need. But the question remains – what are they gaining experience for?

Back to the original question: what is software engineering? It's the ability – through skills, experience and techniques – to organize chaos and create products that are as simple as they can be. That's what a software engineer tries to do – make things as simple. As simple as possible. This takes a lot of experience, luck, and intelligence.

To all my fellow practitioners – hard work, isn't it? And, to my former students – I was doing the best I could!

David A. Cook
Stephen F. Austin State University
cookda@sfasu.edu

P.S. I take no credit for the originality of the toaster story. It's been floating around the web for years.



CIVILIAN TALENT IS MISSION-CRITICAL. LET'S GET TO WORK.



Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to www.navair.navy.mil.

Equal Opportunity Employer | U.S. Citizenship Required



CHOICE IS YOURS.



NAV  AIR



CROSSTALK thanks the above organizations for providing their support.