CarnegieMellon
Software Engineering Institute

news @ sei
interactive

Volume 4 | Number 1 | First Quarter 2001

http://www.interactive.sei.cmu.edu

news @ sei

interactive

From the Director

In the January 2001 issue of *Computer* magazine, Barry Boehm and Vic Basili present a "Software Defect Reduction Top 10 List."[1] I commend the article to you. Let me repeat five items from the list:

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design stage.

2. Current software projects spend about 40 to 50 percent of their time on avoidable rework.

3. Peer reviews catch 60 percent of the defects.

4. Disciplined personal practices can reduce defect introduction rates by up to 75 percent.

5. About 40 to 50 percent of user programs contain non-trivial defects.

Sadly, these are not new findings. They reflect what we have known for many years. As software engineers, we and our management know what to do to improve our software. We just often choose not to do it.

This situation may be made worse by legislation being considered by many state legislatures. The Uniform Computer Information Transactions Act (UCITA), which has already been passed in the Maryland and Virginia legislatures, says, in part, that vendors of software cannot be held liable for the defects in the products that they produce. Proponents of UCITA argue that software is becoming a service rather than a manufactured good, that zero-defect software is difficult if not impossible to produce, and that software products should not be held to the same quality standards as are more tangible products. For more information about UCITA, see http://www.badsoftware.com.

The SEI, as well as professional societies such as the Institute for Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM), have taken positions against UCITA. (For a lengthier treatment of the problem of software quality, please see an article that I coauthored with SEI Fellow Watts S. Humphrey for a recent hearing held by the Federal Trade Commission on UCITA. The article is at http://www.ftc.gov/bcp/workshops/warranty/comments/crossstephene.pdf.)

---

[1] Boehm, Barry, & Basili, Victor R. "Software Defect Reduction Top 10 List." *Computer*, January 2001, pp. 135-137.

Software engineering practices exist that enable software developers to produce near-zero-defect software. Our Web site provides data and case studies to prove this point (see, for example, http://www.sei.cmu.edu/tsp/results.html). The problem is that these best practices in software engineering are not being adopted and used as broadly as they could be.

For an expanded discussion of this topic, including the SEI's list of four key areas where software engineers can improve, please see the article "Pursue Better Software, Not Absolution for Defective Products" in this issue of news@sei interactive.

Also in this issue, we want to tell you about several exciting developments that will help improve the state of software engineering practice. The first article, "Finding Errors Using Model-Based Verification," updates you on progress in model-based verification, a technology that specifically addresses "non-trivial defects" in user programs (item 5 from the Boehm and Basili list). The second, "Intrusion Detection Systems," describes the importance and uses of intrusion-detection systems in your organization's network security plan. The third article, "Improving Technology Adoption Using INTRo," provides insights on how to introduce new and improved practices and technology into organizations. The final article, "Advancing the State of Software Product Line Practice," is based on the work of leading practitioners who have found ways to leverage their investment in producing reusable, high-quality software.

Our columns in this issue cover methods for working successfully with software vendors (The COTS Spot), ways that the FBI investigates computer crime (Security Matters), and architecture mechanisms—ensembles consisting of a few component and connector types (The Architect). We also present the first column in a new series on the future of software engineering by Watts Humphrey (Watts New).

Stephen E. Cross
Director, Software Engineering Institute

Pursue Better Software, Not Absolution for Defective Products

Avoiding the "Four Deadly Sins of Software Development" Is a Good Start

Should software producers be absolved from any problems caused by their defective products? Some software makers think so, and their allies in state and federal governments would like this absolution to be written into law.

The Uniform Computer Information Transactions Act (UCITA), which has already been passed in the Maryland and Virginia legislatures, says in part that vendors of software cannot be held liable for defects in the products they produce.

The Software Engineering Institute (SEI) strongly opposes UCITA, as do prestigious professional societies, including the Institute for Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). (For more information about UCITA, see http://www.badsoftware.com.)

False Assumptions

We at the SEI believe that the rationale for granting special treatment to the software industry relies on four false assumptions:

1. High-quality software cannot be produced. This is demonstrably incorrect.

2. It costs too much to produce defect-free software. This is also demonstrably incorrect.

3. Customers do not want software that is defect free. This is the same argument that U.S. auto manufacturers used before Japanese cars began to take a big bite out of the U.S. market share—because people did value reliability when they had a choice. Why shouldn't the same be true for software? As the software market matures, will India, with its growing number of Capability Maturity Model® Level 5 companies, become the source of software that people will prefer to buy?

4. *Software is a vital industry that should be protected.* We agree that the software industry is vital. But it is not in the national interest to protect an industry that persists in using undisciplined and poor-quality practices, particularly when high-quality software can be produced at an acceptable cost (see assumptions 1 and 2).

Dispelling Myths About Software Quality

Two recent publications have sought to dispel the myth that high quality is impossible by presenting specific guidelines and techniques for enhancing software systems:

- "How to Eliminate the Ten Most Critical Internet Security Threats: The Experts' Consensus," published by the SANS Institute at http://www.sans.org/topten.htm. The list, compiled by representatives from industry, government, and academia, including the SEI's CERT® Coordination Center (CERT/CC), can help administrators harden their systems against the handful of software vulnerabilities that account for the majority of successful attacks. The SANS Institute Web site also includes a step-by-step tutorial.

- "Software Defect Reduction Top 10 List," by Barry Boehm and Victor R. Basili, published in the January 2001 issue of the IEEE journal *Computer*. Boehm and Basili provide an update of their 1987 article "Industrial Metrics Top 10 List" with techniques that can help developers reduce flaws in their code.

Software Defect Reduction
Top 10 List

By Barry Boehm and Victor R. Basili

from IEEE Computer, January 2001, p. 135-137.

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.

2. Current software projects spend about 40 to 50 percent of their effort on avoidable rework.

3. About 80 percent of avoidable rework comes from 20 percent of the defects.

4. About 80 percent of the defects come from 20 percent of the modules, and about half the modules are defect free.

5. About 90 percent of the downtime comes from, at most, 10 percent of the defects.

6. Peer reviews catch 60 percent of the defects.

7. Perspective-based reviews catch 35 percent more defects than nondirected reviews.

8. Disciplined personal practices can reduce defect introduction rates by up to 75 percent.

9. All other things being equal, it costs 50 percent more per source instruction to develop high-dependability software products than to develop low-dependability software products. However, the investment is more than worth it if the project involves significant operations and maintenance costs.

10. About 40 to 50 percent of user programs contain nontrivial defects.

The Four Deadly Sins of Software Development

In this issue we present our own list, albeit one with a somewhat broader perspective. We hope the software community will consider, and build upon, the following Four Deadly Sins of Software Development.

The First Deadly Sin: Sloth
Who Needs Discipline?

A large number of defects are introduced into a typical software product from the detailed design phase through the unit test phase. Finding and fixing defects through testing, which is the usual method, can consume a great deal of time—often up to half the total development time. Furthermore, testing only uncovers some of the defects; 50% is generally considered to be a good number in the software industry. So even after organizations spend considerable resources in software testing, the product that comes out of testing is not defect free. To get defect-free products out of test, organizations must put essentially defect-free products into test.

Boehm and Basili point out that "disciplined personal practices can reduce defect introduction rates up to 75%." By applying the principles of the Personal Software Process[SM] (PSP[SM]) and the Team Software Process[SM] (TSP[SM]), developed at the SEI, or the Cleanroom software development process, originally developed by Harlan Mills, organizations can greatly reduce the number of defects present when the software reaches the testing phase, reduce the number of defects introduced into the software, and can achieve near-zero defects in delivered software.

The PSP method, for example, helps engineers improve by bringing discipline to the way they develop software. The TSP approach is designed to transition those disciplined methods into an organization, by helping its members form and manage high-performance teams. Together, PSP and TSP offer organizations the ability to produce very high quality software, to minimize schedule variance, and to improve cycle times.

In the PSP classes taught at the SEI and by SEI-licensed instructors, engineers learn sound design and verification practices, as well as structured and measured review procedures. SEI data shows that at the start of PSP training, students inject 10 defects in every 100 lines of code they write. Using PSP techniques, this number drops to about 0.7 defects per 100 lines of code. This is data from classroom projects, but real-world analysis shows similar results.

PSP- and TSP-trained engineers also catch defects earlier in the process, says the SEI's Noopur Davis. "Because early defect removal requires much less effort than late defect removal, and because there are fewer defects to find and fix, the cycle time decreases and quality increases." In other words, these practices can help engineers overcome another of Boehm and Basili's findings: "Current software projects spend about 40 to 50% of their effort on avoidable rework."

This approach differs significantly from the method for finding defects through testing, which generally involves a time-consuming process of looking at the symptoms of a problem—say, a system crash—and working backwards. Davis says, "You have to ask, 'how could I have arrived at this point in the program?' and you have to look at all the paths you could have taken to get there. Very often, the symptom has very little to do with the cause of the problem."

The Second Deadly Sin: Complacency
The World Will Cooperate with My Expectations

This sin can be the most lethal when developers are so complacent that they fail to defend against hostile input. Eighty-five percent of all security problems can be placed in the category of "failing to defend against hostile input," says Shawn Hernan of the CERT/CC. What makes this sin especially egregious, he says, is that in many cases the defense would be easy to achieve: simply avoid software development mistakes that have already been identified and are well known, or make the necessary repairs, such as those recommended by the CERT/CC at its Web site (http://www.cert.org) and those listed on the SANS Institute's top 10 list. "Most problems involve the same mistakes committed over and over," Hernan says.

The training that programmers receive is partly to blame, Hernan says. When software developers first learn to program, they are told to focus on functionality and performance—for example, how to make a program do what it's supposed to do faster. "But little attention is paid to making sure a program doesn't do what it's not supposed to do, when its assumptions are violated."

Programmers also unwittingly open the door to hostile input when they try to extend the functionality of one program into a domain for which it was not intended. "You want to reuse everything that you can, but you have to be careful how you do it," Hernan says. "You must understand the assumptions of the developer and packager before you reuse something in a system that has security requirements."

One example of failing to defend against hostile input is the CGI program "phf," a directory service that was once included with popular server software. The program provided a Web-based form that was designed to allow users to get legitimate

information from a directory, such as telephone numbers ("ph" stands for "phone" and "f" stands for "form"). The problem was that the easy-to-use form would construct a UNIX command line and execute it exactly as if it were typed into an actual command line—in other words, it gave users of the Web form command-line access to the server. Instead of simply typing in a name on the form to get a telephone number, an intruder could plug in, for example, "$ph smith; cat /etc/passwd" and gain access to the password file on the server. "The problem comes in using a system command over the Web. The system command is way too powerful for this purpose," Hernan says.

The implications of allowing hostile input into a computer system are myriad and extreme, and can include a compromise of an entire system. In the case of phf, the CERT/CC received hundreds of reports of hostile users exploiting the security flaw to modify Web sites and compromise computers.

The CERT/CC advises writers of software code to include checks of all input for size and type to make sure it is appropriate. This prevents the use of special characters, which was one of the problems with phf. The CERT/CC also advises code writers to limit inputs to the allocated space, which helps prevent buffer overflows that could occur when an intruder attempts to store more data in a buffer than it can handle, resulting in the loss of service on a computer or system. The SEI technical report rlogin(1): The Untold Story, provides an analysis of software defects that could result in buffer overflows from one well-known case, and some general guidelines and mitigation strategies for preventing similar defects.

Hernan points out that developers should be especially cautious when dealing with Web-based programs, which should be considered "privileged" because they can grant extraordinary access to users. "Anything that operates over the network is a privileged program. So, anything that operates over the Web is a privileged program. If you extend a privilege to people everywhere, then you give them the ability to run anything on your system."

The Third Deadly Sin: Meagerness
Who Needs an Architecture?

"An explicit software architecture is one of the most important software engineering artifacts to create, analyze, and maintain," says the SEI's Mark Klein. Yet, developers often leave the architecture invisible, or "it is covered and permuted by the sands of change."

Even when the principles of an intended architecture are laid out in advance, it is difficult to remain faithful to that architecture as design and implementation proceeds. This is particularly true in cases where the intended architecture is not completely specified,

documented, or disseminated to all of the project members. In the SEI's experience, well specified, documented, disseminated, and controlled architectures are the exception. This problem is exacerbated during maintenance and evolutionary development, as the developed product begins to drift from the architecture that was initially intended.

When the architecture of a software system is invisible, it means no one has intellectual control of the system, the design of the system is very difficult to communicate to others, and the behavior of the system is hard to predict. As a result, changes are difficult and costly, and often result in unforeseen side effects.

Suppose that a developer inserts a new middleware layer of software so that distribution and porting of the system become much easier. Without an explicit architecture to refer to, that developer might fail to realize that this change negatively affects the performance of the timing-critical aspects of the system.

An invisible software architecture also represents a missed opportunity. The ability to identify the architecture of an existing system that has successfully met its quality goals fosters reuse of the architecture in systems with similar goals. Indeed, architectural reuse is the cornerstone practice of software product line development.

Developers must build systems with explicit architectures in mind, and must remain faithful to those intended architectures. At the very least, significant changes to a software system must be explicitly recorded. By using techniques such as the Architecture Tradeoff Analysis Method^{SM} (ATAM^{SM}) developers can analyze the tradeoffs inherent in architectural design, so that they can understand how various quality attributes—such as modifiability and reliability—interact. They can then weigh alternatives in light of system requirements.

For systems that are already built, all is not lost. Software architectures can be reconstructed through the acquisition of architectural constraints and patterns that capture the fundamental elements of the architecture. Commercial and research tools are available that provide the basic mechanisms for extracting and fusing multiple views of a software program or system. (The SEI has collected several of these tools and techniques into the "Dali" workbench.) Using such tools can allow for the reconstruction of a system's architecture even when there is a complete lack of pre-existing architectural information, as might be the case when the system being analyzed is particularly old, and thus there are no longer any designers or developers who can relate architectural information.

The Fourth Deadly Sin: Ignorance
What I Don't Know Doesn't Matter

Here, designers develop software systems without showing proper regard for the intended purpose of software components. This problem appears more frequently today as designers increasingly assemble systems by integrating commercial off-the-shelf (COTS) components. To do this properly, designers must fully understand the components they are using. The final system will often fail to meet expectations if designers do not thoroughly investigate the components they select and the tradeoff decisions that must be made and quantified in terms of cost and risk.

The SEI is developing methods that designers can use to make objective evaluations of components and then competently use those components when assembling systems. The goal is to provide owners and users with the system they think they are getting.

"A designer's perceived reality of a product is built up over time and could come from any number of sources." says the SEI's Scott Hissam. "These sources might include product literature, articles in trade magazines, or hearsay." All of these sources suggest to designers what the product can do, such as providing distributed transaction services, load balancing, fault tolerance, or seamless integration of X to Y.

When system designers' perceptions of a component don't match the reality of the product, those designers can expect to run into problems when they try to integrate the component into a system. For example, the product could turn out to have unanticipated security vulnerabilities. Or a designer could find that two components that purport to have the capability to seamlessly integrate do not do so. This might result in a great deal of additional work to make the components function together, or it could mean selecting other components for the overall system, or the overall system, when built, may not meet performance, security, or dependability requirements.

Eliminating Mismatches

To gain the insight needed to make better decisions about component selection, designers must work through their expectations iteratively and learn precisely what the product does, what it requires, and how it behaves. Designers should ask questions such as: "What does the product need from me?" "What does the product need from other products in my environment?" "How does the product behave when I try to use it?" "What resources does it use?" Some intangible aspects of a product might not appear on the license agreement or in the product specification, such as the threading model, latencies, and security (or lack thereof). Such information can often only be gained through actual interaction with the product.

Designers often have "a slightly skewed vision of what the specifications are for products because there is such a huge implied environment that people never document," says Bill Fithen of the SEI's Survivable Systems initiative. "They just assume that, for example, 'UNIX' is a satisfactory definition of the environment in which the program is going to run. Perhaps 90% of the errors that we see fall into that category: the designers did not consider what the environment really looked like. They considered some aspect of it but not all of it."

A pervasive example of undocumented assumptions can be found in the TCP/IP protocols—the suite of communications protocols used to connect hosts on the Internet. TCP/IP was developed by the U.S. Department of Defense to connect a number different networks designed by different vendors into a "network of networks." One of the assumptions made by the originators of the protocol was that parties on both ends of the communication actually wanted to communicate. It never occurred to them that on one end of the communication the goal could be, for example, to gather sensitive data from the system on the other end.

Conclusion

We believe that the practice of software engineering is sufficiently mature to enable the routine production of near-zero-defect software. Our Web site (www.sei.cmu.edu) provides data and case studies to prove this (see, for example, http://www.sei.cmu.edu/tsp/results.html). As consumers of commercial software, we should demand and expect the highest possible quality.

As the U.S. Department of Defense comes increasingly to rely on the commercial software industry to provide components in its systems, it is important to our national defense that software developers be held to standards of quality and accountability that are in effect in other industries.

Ironically, it is in the U.S. software industry's best interests to produce better software, as the SEI's Watts Humphrey has written in a recent column in news@sei interactive (http://interactive.sei.cmu.edu/news@sei/columns/watts_new/2001/ 1q01/watts-1q01.htm). Improved software quality "would mean increased opportunities for computing systems and increased demand for the suppliers' products," Humphrey says.

What's more, continuing to ignore quality—and even to seek legal protection for defective software through such mechanisms as UCITA—will ultimately invite competition from other nations, whose software makers will learn to make superior products and could eventually replace the United States as the industry leader.

We welcome your feedback on this article and invite you, as members of the software-development community, to help us identify more sins of software-development. We plan to update and add to this list at least annually and perhaps more often, depending on your response. Please send your comments and suggestions to interactive@sei.cmu.edu and join us in the quest for the highest possible software quality.

http://interactive.sei.cmu.edu

Finding Errors Using Model-Based Verification

Bob Lang

Formal methods have long offered the promise of ensuring high quality software using mathematical rigor. The director's message in the Spring 2001 issue of news@sei (http://www-preview.sei.cmu.edu/products/news.sei/) points to one article suggesting that 40 to 50 percent of programs contain nontrivial defects. Formal methods represent a clear attempt to address such concerns. However, applying traditional formal methods to a complete system design requires a significant investment–from learning a difficult technology to applying it in all phases of the development effort. As a result, there have been relatively few success stories, and formal methods have failed to achieve widespread adoption.

The SEI has leveraged the work of the formal methods community to develop a software engineering practice known as model-based verification (MBV). MBV involves the selective use of formal methods on abstract models of important portions of a system, thereby providing many of the benefits promised by formal methods without the associated high cost.

About MBV

Model-based verification makes use of "light weight" formal methods and can be used throughout the software life cycle. For example, MBV can be used to augment the standard peer review process to find subtle errors that would not typically be found in such a process. The term "light weight" is used to underscore the fact that MBV relies on selective, focused use of formalism as well as mathematically based techniques not generally classified as formal methods.

MBV makes use of existing techniques such as model checking, in which abstract models of a system are built to capture the salient features of the system or a portion of the system that is considered important. For example, a traffic signal system at an intersection could be modeled at any number of levels: "You may want to model each of the four lights as it goes from green to yellow to red and how they interact with each other," says Chuck Weinstock of the SEI. "Or you may only care that the lights align so that the traffic going one way is stopped when the traffic going perpendicular is moving. Or you may only care that a single light works." MBV allows you to select the abstraction level and then apply the appropriate methods to it.

Once the model is built, it is tested against claims, also written in a formal language. In the traffic signal example, these claims might be the following: "conflicting directions will never be green at the same time," or "at least one direction will always be green." A

tool known as a model verifier would then be used to determine whether the claims are satisfied. "If the model has been correctly built," says Weinstock, "when a negative result comes back, you should be able to go back to the code and find the error."

Benefits of MBV

A key benefit of MBV is that it allows software engineers to find subtle errors and to find them earlier in the life cycle when they are cheaper to fix. "We believe that if MBV reveals even one very subtle error that was not found otherwise," says Weinstock, "it will more than pay for itself." The models also serve as system documentation that otherwise would not exist. Because the model is precise at the abstract level, the documentation can provide a great deal of precision about how the system is supposed to behave. A better understanding of the system allows for easier life cycle support.

In addition, MBV is built on established techniques. Model checking is used routinely in complex hardware design and has been used successfully in protocol design. For example, when one organization wanted to test its new model checking tool, it tested the tool on a new product that it considered complete and ready for release. The product was seeded with a few errors and the tool was applied to see whether the tool would find the errors. The tool found not only the errors that were seeded, but additional errors that would have required the system to be recalled if it had been released.

As noted above, formal methods can be expensive to apply. "Furthermore," says Weinstock, "it typically takes people with highly specialized knowledge to use formal methods. By contrast, the basics of MBV can be learned by a good software engineer in a week or so of training and study."

Putting MBV into Practice

SEI staff are currently codifying MBV into a software engineering practice and transitioning the practice to the software community through the use of courses, handbooks, and pilot studies. A half-day version of an MBV tutorial was given at the SEI Software Engineering Symposium in September 2000. A full-day version is being developed and a draft MBV handbook is in progress.

A pilot study is currently being conducted with NASA, and involves the X33 Reusable Launch Vehicle. The information gathered will help to refine MBV, making it more suitable for broad adoption in the software engineering community.

Intrusion Detection Systems

Intrusion Detection Systems (IDSs) are an important, evolving technology for protecting computer networks. All sectors of society, from government to business to universities, increasingly depend on networks to be reliable and secure. For many e-commerce organizations, their very existence in the marketplace demands expert computer security practices. For instance, in the 1980s and early 1990s, denial-of-service (DoS) attacks—a kind of attack that floods the network until it can no longer handle legitimate traffic—were infrequent and not considered serious. Today, successful DoS attacks can cause great financial loss to organizations such as online stock-brokers and retailers and can cause a wide array of unwanted disruptions to government computer networks. (For more on denial of service attacks, see the Spring 2000 issue of news@sei available at http://www.sei.cmu.edu/products/news.sei/news-spring00.pdf.)

The goal of intrusion detection systems is to accurately detect anomalous network behavior or misuse of resources, sort out true attacks from false alarms, and notify network administrators of the activity. Many organizations now use intrusion detection systems to help them determine if their systems have been compromised, but it can be difficult to find unbiased information to help organizations understand and evaluate the available tools and how to best use them.

Although intrusion detection and response technology is still immature, it remains a key element in a layered, enterprise-wide security plan that often includes detailed security policies, a computer security incident response team (CSIRT), firewalls, virus protection, encryption, authentication, access control, virtual private networks (VPNs), and more. IDSs are essential because, at best, they provide increased near real-time detection that can help limit compromise and damage to networks, reduce costs through automated detection, and stem DoS attacks.

The CERT® Coordination Center (CERT/CC) at the SEI publishes security improvement modules (each of which addresses an important but narrowly defined problem in network security) and technical reports, and offers courses that contain intrusion detection information. The technical report *State of the Practice of Intrusion Detection Technologies* offers detailed guidance about all phases of selecting, deploying, and maintaining IDSs. The following sections outline some important concepts from CERT/CC publications.

Selecting an IDS

When selecting an IDS, organizations need to consider the level of privacy needed, how much the organization can afford to spend, and whether there are internal constraints on

the types of software the organization can use. Other important topics include specifics about IDS capabilities, such as detection and response characteristics, use of signature and/or anomaly-based approaches, accuracy of diagnosis, ease of use, and effectiveness of the user interface.

Since the new product cycle for commercial IDSs is rapid, information and systems quickly become obsolete. Staff of the CERT/CC have conducted experiments with commercial and public domain tools, and found that commercial IDS tools were easier to install than public domain tools, but neither had an understandable, easy-to-use, configuration interface. The majority of IDSs provide good capabilities for enhanced network monitoring rather than for intrusion detection, given that some post-IDS-alert data analysis was normally required. In many cases, determining which features are most important to an organization will be the deciding factor.

Although an IDS is an important element in an organization's overall security plan, it is only effective if it has support from management at the level of the corporate chief information officer and the information security manager. They must ensure that the IDS is properly deployed and maintained.

Deploying an IDS

Once an IDS is selected, a number of decisions will determine whether it is deployed effectively. These include decisions about how to protect the organization's most critical assets, how to configure the IDS to reflect the organization's security policies, and what procedures to follow in case of an attack to preserve evidence for possible prosecutions. Organizations must also decide how to handle alerts from the IDS and how these alerts will be correlated with other information such as system or application logs.

Additional information and guidance in selecting an IDS are available from SEI publications, including:

State of the Practice of Intrusion Detection Technologies
http://www.sei.cmu.edu/publications/
documents/99.reports/
99tr028/99tr028abstract.html

Preparing to Detect Intrusions
http://www.cert.org/
security-improvement/modules/m09.html

Responding to Intrusions
http://www.cert.org/
security-improvement/modules/m06.html

An IDS does not prevent attacks. In fact, if attackers realize that the network they are attacking has an IDS, they may attack the IDS first to disable it or force it to provide false information that distracts security personnel from the actual attack. Many intrusion detection tools have security weaknesses that could include failing to encrypt log files, omitting access control, and failing to perform integrity checks on IDS files.

Maintaining an IDS

An IDS must be constantly monitored after it is deployed. Procedures must be developed for responding to alerts; these procedures will determine how staff members analyze and act on alerts, and how staff monitors the outcomes of both manual and automatic responses. In addition, as upgrades become available, they should be installed to keep the IDS as current and secure as possible.

Technology alone cannot maintain network security; trained technical staff are needed to operate and maintain the technology. Unfortunately, the demand for qualified intrusion analysts and system/network administrators who are knowledgeable about and experienced in computer security is increasing more rapidly than the supply.

When an IDS is properly maintained, it can provide warnings about when a system is being attacked, even if the system is not vulnerable to the specific attack. The information from these warnings can be used to further increase the system's resistance to attacks. An IDS can also confirm whether other security mechanisms, such as firewalls, are secure. If the necessary time and effort is spent on an IDS through its life cycle, its capabilities will make it a useful and effective component of an overall security plan.

Improving Technology Adoption Using INTRo

Lauren Heinz

As software-intensive organizations increasingly recognize the need for guidance in seamlessly introducing new software tools and technologies, many practitioners are discovering they lack the skills necessary to champion successful adoptions.

To help organizations systematically analyze, select, and implement new technology, the SEI in collaboration with Computer Associates has developed a high-level approach for technology implementation known as the IDEAL[SM]-Based New Technology Rollout (INTRo). INTRo is a Web-based resource that guides software professionals through technology change management through organizational and technological assessments, proven practices, and methodologies that foster learning.

From IDEAL to INTRo

INTRo builds from IDEAL, a comprehensive approach to long-term software improvement through the five key process stages for which it is named: Initiating, Diagnosing, Establishing, Acting, and Learning. IDEAL serves as a model for planning and implementing effective improvements. "We wanted to take advantage of the lessons learned from using IDEAL in software process improvement, and to extend and apply those lessons to the new domain of in-formation technology (IT) and commercial off-the-shelf (COTS) software selection and deployment," says Linda Levine of the SEI. "INTRo goes further by providing a greater level of detail, content, and support for users."

Using INTRo, an organization can introduce new software tools and technologies by following a series of structured and informative process steps, tutorials, tips, checklists, and sample process outputs. The model emphasizes the importance of sharing information and disseminating knowledge practices throughout an organization in order to develop more lasting and complete business solutions. Further, INTRo addresses change across the dimensions of process, technology, people, and culture. While typical change models tend to focus on one or two dimensions, INTRo provides a more customized method across multiple variables.

As interactive learning and collaborative practice were integral to developing INTRo, Levine hopes to stimulate an online community around the use of INTRo by developing a Web-based forum for users to share lessons and explore alternative approaches to technology adoption. "Such a forum for exploration and learning would allow the SEI to

extend its interactions with its partners and constituency and to build a community of practice," Levine says.

INTRo is designed to meet the needs of large efforts, including multi-site, cross-organizational, enterprise-wide rollouts. INTRo is most applicable to organizations at maturity level 3 and higher on the Capability Maturity Model® for Software (SW-CMM®). For small projects, such as a development team of a few engineers adopting a new tool, INTRo may be applied informally or in part. In the case of such informal and partial use, adoption results will vary.

The INTRo Process

IDEAL-Based New Technology Rollout consists of seven stages: Project Initiation, Problem/Domain Analysis, Solution Definition, Technology Customization and Testing, Whole-Product Design, Breakthrough, and Rollout.

- Project Initiation establishes project and management goals, and structures the effort. After defining a project's key players, scope, purpose, resources, time variables, deliverables, benefits, and work processes, the project team develops these ideas into a formal, resourced, and funded project.

- Problem/Domain Analysis analyzes the connection between the business problem and the new technology area, assesses the current environment, and develops user requirements. In the documentation stage, the team assesses its organization's strengths and weaknesses in the technology area, analyzes improvement opportunities, drafts a new business process, and develops a prototype of a "whole-product" solution, as described in the whole product design stage below.

- Solution Definition identifies two or more of the solution options, and plans for later implementation. The solution is selected based on its ability to support the technical and business requirements and cost effectiveness. Team members identify and evaluate component packages, recommend product purchases, and conduct early test planning.

- The purpose of Technology Customization and Testing is to adapt the core technology; design and perform data migration; design, develop, and execute tests; and perform integration. The Technology Solution and Desired State products are used to guide the selection team to products that will achieve the business goals/drivers for the architecture.

- Whole-Product Design consists of planning and developing each whole-product component so that when the Breakthrough stage begins, the whole-product solution will be ready to be implemented. The whole-product solution is concerned with introducing the new technology with a level of service that includes the user's perception of that service. During this stage, the project team maps out the following components:

- social design (organizational structure, rewards and incentives, performance measures)

- policies and standards

- support mechanisms

- training

- knowledge and skills transfer mechanisms

- In Breakthrough, the team pi-lots the solution to evaluate and fine tune it before full deployment. Breakthrough is also an important marketing opportunity; users throughout the organization will ask team members what they think about the new technology and how it is being introduced. Difficulties in Break-through can jeopardize the success of Rollout. The goal is to determine the most predictable introduction process for the new technology solution with the least impact on productivity, cycle time, and quality.

- Rollout moves the solution out to all the business units that are adopting the new technology and implements it throughout the organization. The team builds an iterative rollout plan, and purchases technical components. Rollout includes briefing and training users, installing and testing tools, and activating the whole-product solution. When this stage is complete, the technology solution is periodically analyzed and validated for product and process improvement.

INTRo is available for pilots or early customer use. Plans are currently being made for additional pilots. Please contact us if you are interested in being an early adopter.

http://interactive.sei.cmu.edu

Advancing the State of Software Product Line Practice

Organizations developing software-intensive systems face a number of challenges. These include long product cycles, very little return on investment from software assets, and difficulty with systems integration.

A product line approach for software can overcome these challenges. A software product line is a set of products sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Software product lines amortize the investment in core assets through strategic reuse. Organizations that apply this approach can develop and maintain multiple software products quickly, efficiently, and economically. Furthermore, software product lines can improve product quality and reduce system integration costs.

Currently, a large number of organizations are using a product line approach for their software and achieving large-scale productivity gains, reduced time to market, and increased customer satisfaction. However, there remain many organizations that require the benefits of the strategic, large-grained reuse characteristic of software product lines, but do not know where, when, or how to start.

The SEI is helping these organizations by distilling, codifying, tailoring, and transitioning the technology organizations need to successfully implement software product lines. At the same time, the SEI is helping to nurture a software product line community. To that end, the SEI organized and sponsored the first major international conference devoted exclusively to software product lines.

The First Software Product Line Conference (SPLC1)

SPLC1 brought together leaders in software product lines from industry, academia, and the government. According to Conference Chair Linda Northrop, the SEI had several goals for the gathering: "The SEI organized this event to bring together the emerging software product line community. We also wanted to provide a forum for the exchange of ideas and information related to current research and practice."

SPLC1 attracted 185 participants from North America, Europe, Asia, Africa, South America, and Australia. Recognized software product line authorities from Rational, Lucent Bell Labs, the European Software Institute, Raytheon, the University of Texas at

Austin, Hewlett-Packard, Thomson-CSF, Siemens, Robert Bosch Corporation, Philips, Boeing, and other well-regarded organizations made up the program committee.

Committee members reviewed a total of 59 submissions from which they selected 27 technical papers for presentation. The technical papers reported both software product line research and experience. The resultant technical paper sessions covered a broad range of product line topics: practice and experience, organization and management, methods, process, components, architecture, tools and techniques, and domain engineering.

In addition to the technical paper sessions, SPLC1 included ten tutorials, seven workshops, two panels, and a keynote lecture, all on a broad range of software product line technology and issues. The conference also initiated a Software Product Line Hall of Fame to support and recognize excellence in software product lines. SPLC1 participants nominated those software product lines that serve as exemplary models, that were designed as product lines and paid explicit attention to commonality and variation, and that were a commercial success. Four product lines were selected, the U.S. Navy's A7 Avionics System, CelsiusTech SS2000 command and control system, HP Owen Printer product line, and Nokia mobile cellular phone product line.

Conference proceedings, *Software Product Lines, Experience and Research Directions*, were published as part of the Kluwer International Series in Engineering and Computer Science.[1]

The SPLC1 brought the global software product line community together for four days of vibrant information exchange in a refereed forum. A digest of SPLC1 can be found at http://www.sei.cmu.edu/plp/conf/SPLC.html. To build on this foundation, the SEI is planning the Second Software Product Line Conference (SPLC2) for 2002. The SEI is currently soliciting input for SPLC2. To participate, go to http://www.sei.cmu.edu/plp/SPLC2_questionaire.html.

Framework 3.0 Released

As part of its product line practice work, the SEI seeks to mature product line technology and disseminate it to the community. The publication of *A Framework for Product Line Practice Version 3.0* represents an important milestone in this effort.

---

[1]   Donohoe, Patrick, ed. *Software Product Lines, Experience and Research Directions*. Boston, MA: Kluwer Academic Publishers, 2000.

A Framework for Software Product Line Practice is a constantly evolving, Web-based document. The SEI developed the framework to

- identify the concepts underlying software product lines and the essential activities to consider before creating or acquiring a software product line

- identify practice areas that an organization creating or acquiring software product lines must master

- define practices in each area, where current knowledge is sufficient

- guide an organization through the process of moving to a product line approach for software

The framework is available online at http:/www.sei.cmu.edu/plp/framework.html.

To date, 29 practice areas required for product lines have been defined. This information has been culled from research, workshops, product line collaborations with customers, and feedback from the community. Over the past years, the framework has been used in scores of organizations in their software product line efforts. Version 3.0 represents a significant advance over Version 2.0 in that all practice areas have been completely defined. The framework will continue to be improved as more product line information is gathered.

The SEI is also developing *An Acquisition Companion to the Framework for Software Product Line Practice*. Designed for the DoD acquisition community, this document presents information to streamline the process of commissioning and implementing software product lines within a DoD environment.

In addition, a book based upon the SEI's work, *Software Product Lines: Practices and Patterns*, will be published this summer by Addison-Wesley as part of the Addison-Wesley SEI Series in Software Engineering.

http://interactive.sei.cmu.edu

Architecture Mechanisms

Len Bass, Mark Klein, Rick Kazman

Architecture mechanisms—ensembles consisting of a small number of component and connector types—allow designers to explore and document relationships between software architectural styles and quality attributes. This enables designers to identify choices they must make in the pursuit of quality attribute goals. Study of these mechanisms can also help set a foundation for further software architecture design and analysis.

Introduction

Over the past 30 years, a number of researchers and practitioners have examined how systems affect software quality attributes. However, no one has systematically and completely documented the relationship between software architecture and quality attributes.

The lack of documentation stems from several causes:

- The lack of a precise definition for many quality attributes inhibits the exploration process. While attributes such as reliability, availability and performance have been studied for years and have generally accepted definitions, other attributes, such as modifiability, security, and usability, do not have generally accepted definitions.

- Attributes are not discrete or isolated. For example, availability is an attribute in its own right. However, it is also a subset of security (because it can be affected by denial-of-service attacks) and usability (because users require maximum uptime). Is portability a subset of modifiability or an attribute in its own right? Both relationships exist in quality attribute taxonomies.

- Attribute analysis does not lend itself to standardization. There are hundreds of patterns at different levels of granularity, with different relationships among them. As a result, it is difficult to decide which situations or patterns to analyze for what quality, much less to categorize and store that information for reuse.

- Analysis techniques are specific to a particular attribute. Therefore, it is difficult to understand how the various attribute-specific analyses interact.

Notwithstanding the difficulties, systematically codifying the relationship between architecture and quality attributes has several obvious benefits, including:

- a greatly enhanced design process—for both generation and analysis

- during design generation, a designer could reuse existing analyses and determine tradeoffs explicitly rather than on an ad hoc basis. Experienced designers do this intuitively but even they would benefit from codified experience. For example, during

analysis, designers could recognize a codified structure and immediately know its impact on quality attributes.

- a method for manually or dynamically reconfiguring architectures to provide specified levels of a quality attribute. Understanding the impact of quality attributes on architectural mechanisms will enable designers to replace one set of mechanisms for another when necessary.

- the potential for third-party certification of components and component frameworks. Once the relationship between architecture and quality attributes is codified, it is possible to construct a testing protocol that will enable third-party certification.

For all these reasons, the idea of documenting the relationship between architecture and quality attributes is a worthy goal.

A universally accepted premise of the software architecture community is that architecture determines attributes. This raises several questions:

- Is this premise true and why do we believe that quality attribute behavior and architecture are so intrinsically related?

- Can we pinpoint key architectural decisions that affect specific quality attributes?

- Can we understand how architectural decisions serve as focal points for tradeoffs between several quality attributes?

The concept of an architecture mechanism allows us to address these questions. An architecture mechanism is an ensemble consisting of a small number of component and connector types. It significantly affects quality attribute behavior and has sufficient content for analysis.

For example, encapsulation is a mechanism for achieving modifiability. Replication is a mechanism for achieving reliability. We contend that these architectural mechanisms are design primitives. They allow designers to explore and document the relationship between software architectural styles and quality attributes. At this point, we believe that there are fewer than 10 mechanism types per quality attribute.

By codifying mechanisms, designers can also identify the choices necessary for achieving quality attribute goals. This, in turn, will set a foundation for further software architecture design and analysis. In fact, this work was motivated by the need for such foundations for use in the Architecture Tradeoff Analysis Method[SM] (ATAM) and the Attribute-Driven Design (ADD) method (http://www.sei.cmu.edu/publications/documents/00.reports/00tr001.html).

Of course, there are many challenges. One challenge is identifying the correct collection of mechanisms at the proper level of abstraction. Another is understanding how to

combine mechanisms into styles and styles into systems, while combining the analyses associated with each.

Architecture Mechanisms

Architecture mechanisms directly contribute to quality attributes. Examples of architecture mechanisms are data routers, caching, and fixed-priority scheduling. These mechanisms help achieve specific quality attribute goals as defined by *general scenarios*:

- Data routers protect producers from additions and changes by consumers, and vice versa, by limiting the knowledge that producers and consumers have of each other. This affects or contributes to modifiability.

- Caching reduces response time by providing a copy of the data close to the function that needs it. This contributes to performance.

- Fixed-priority scheduling allows the interleaving of multiple tasks to control response time. This contributes to performance.

Each mechanism is implemented by a small number of components and connector types. Each is targeted at one or more quality attributes and uses a single strategy for achieving these attributes. For example, the data router uses indirection to achieve modifiability. Caching uses replication to achieve performance. Fixed-priority scheduling uses pre-emptive scheduling to achieve performance.

In essence, architecture mechanisms are the design primitives for achieving quality attribute behavior in a system. An architecture mechanism may be an existing pattern or style. The goal of our work is to identify the specific design primitives that elicit quality attributes, and to then analyze those primitives from the point of view of multiple quality attributes.

For example, a client–server is an architecture mechanism. It supports modifiability by enabling additional clients (up to a point). However, it can also support performance by bringing computation closer to the user.

As you can see, we can easily extend the list of mechanisms and general scenarios to address a wide range of situations. If we introduce a new general scenario, we may need to introduce new mechanisms that contribute to it. Alternatively, we may discover new mechanisms for dealing with existing general scenarios. In either case, we can describe the new mechanism without affecting any other mechanism. This allows us to introduce new design primitives into our analysis incrementally.

Levels of Abstraction for Mechanisms

Mechanisms exist in a hierarchy of abstraction levels. For example, separation is a mechanism that divides two coherent entities. This represents a high level of abstraction. A more concrete mechanism, at a lower level of abstraction, would specify entities, such as separating data from function, function from function, or data from data. Increasing detail also increases the precision of our analysis. However, for purposes of establishing design primitives, our goal is to determine the highest level of abstraction that still supports some useful analysis.

What qualifies as an analysis? Earlier we stated that an architecture mechanism helps achieve quality attribute goals. There must be a reason why the mechanism supports those goals. Articulating the reason becomes the basis for our analysis. If we cannot state why a particular collection of component and connector types supports a quality attribute goal, then the collection does not qualify as a mechanism.

Choosing the correct level of granularity addresses the proliferation of architectural design patterns, and overcomes the third problem preventing the relationship between structure and quality attributes from being codified.

Desired Attribute and Side Effects

Earlier we stated that mechanisms have two consequences: they help particular general scenarios achieve quality attributes and they can influence other general scenarios as a side effect. Given two consequences, we need two types of analysis. The first analysis will explain how the architecture mechanism helped the intended general scenario achieve its result. The second analysis describes how to refine the general scenario in light of the information provided by the mechanism. This analysis reveals side effects the mechanism has on the other general scenarios. By referring to a mechanism that directly affects that attribute, we can perform a more detailed analysis of the side effect.

Let's use encapsulation again as our example. We will analyze the modifiability general scenarios that reflect changes in function, environment, or platform. Our analysis states that as long as changes occur inside the encapsulated function (that is, as long as they are hidden by an interface), the effect of these changes will be limited. If the expected changes cross the encapsulation boundary, the effect of the changes is not limited.

The analysis/refinement for the performance general scenarios asks whether encapsulation significantly affects resource usage of the encapsulated component.

The analysis/refinement for the reliability/availability general scenarios asks whether encapsulation significantly affects the probability of failure of the encapsulated

component. A more detailed description of the performance or reliability analyses would be found in appropriate performance or reliability mechanism writeups.

Figure 1 shows the relationship between mechanisms and general scenarios. If mechanisms are across the top, and general scenarios are down the left side, then each cell has one of three possibilities.

1. the mechanism contributes to this general scenario

2. the mechanism has no affect on the general scenario

3. the mechanism introduces side effects—questions to be answered by other mechanisms

	Mechanism 1					Mechanism N
General Scenario 1	Analysis	Side effect	Does not apply	Side effect	Analysis	Does not apply
	N/A					
	N/A					
	Side effect					
	Side effect					
	N/A					
	N/A					
General Scenario N	Side effect	N/A	N/A	Analysis		

Figure 1: Relationship Between Mechanisms and General Scenarios

When describing the analyses to be performed, we group the general scenarios into attribute headings. Thus, in the mechanism template there are sections entitled modifiability, reliability/availability, performance, security, and usability. One or more of these is the attribute that the mechanism is intended to address. These are listed beside the attributes that are side effects of the mechanism.

Mechanisms and ABASs

Attribute-Based Architectural Styles (ABASs) (http://www.sei.cmu.edu/publications/documents/99.reports/99tr022/99tr022abstract.html) associate classes of architectures with attribute-specific analysis. Architectural mechanisms are quality attribute design primitives. Therefore, we should be able to discuss ABASs in terms of architecture mechanisms. Consider the following examples.

- The publish/subscribe modifiability ABAS consists of the router and registration mechanisms.

- The tri-modular redundancy (TMR) reliability ABAS consists of the (functional) redundancy and (majority) voting mechanisms.

- The Simplex^{TM} reliability ABAS consists of the (analytic) redundancy and (rule-based) voting mechanisms.

While we intend to explore the relationship between ABASs and mechanisms in the future, we can make several observations right now. Both mechanisms and ABASs are attribute-specific design building blocks containing architectural as well as analysis information.

However, ABASs are "bigger" and more concrete than mechanisms. In the above examples, ABASs comprise more than one mechanism. Yet the type of redundancy and voting mechanisms used by TMR are different from those used by Simplex. Mechanisms, by comparison, are intended to be design primitives and should be indivisible.

The Future of Mechanisms

The work on mechanisms at the Software Engineering Institute (SEI) is a natural outgrowth of previous work on the Architecture Tradeoff Analysis Method and ABASs. We will, over the next few months, be publishing a technical note and eventually a larger report describing a large set of architecture mechanisms and detailing their uses.

We are looking for reviewers, contributors, and users from the industrial and governmental architecture community to aid us in the process of creating and validating a useful and used collection of mechanisms.

About the Authors

Len Bass is a senior member of the technical staff at the Software Engineering Institute of Carnegie Mellon University (CMU). He has written or edited six books and numerous papers in a wide variety of areas of computer science including software engineering,

human-computer interaction, databases, operating systems, and theory of computation. His most recent book Software Architecture in Practice (co-authored with Paul Clements and Rick Kazman) received the Software Development Magazine's Productivity Award. He was the primary author of the Serpent User Interface Management System, which was widely cited for its innovativeness and influence. He organized a group that defined a user-interface software reference model that is an industry standard, and headed a group that developed a software architecture for flight training simulators that has been adopted as a standard by the U.S. Air Force. He also headed a group that developed a technique for evaluating software architectures for modifiability. He is currently working on techniques for the analysis of software architectures, and techniques for the development of software architectures for product lines of systems. He is working to understand how to achieve usability through software architecture and has recently been involved in several applications involving mobile computers including wearables and automotive computers. He is the representative of the ACM to the International Federation of Information Processing technical committee on Software: Theory and Practice. Before joining CMU in 1986, he was professor and chair of the Computer Science Department at the University of Rhode Island. He received his Ph.D. in computer science in 1970 from Purdue University.

Mark Klein is a senior member of the technical staff at the SEI, where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He has more than 20 years of experience in research and technology transition on various facets of software engineering and real-time systems. Klein's most recent work focuses on the analysis of software architectures. His work in real-time systems involved the development of rate monotonic analysis (RMA), the extension of the theoretical basis for RMA, and its application to realistic systems. He has co-authored many papers and is a co-author of the RMA Handbook, titled *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-time Systems*.

Rick Kazman is a senior member of the technical staff at the SEI, where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He is also an adjunct professor at the Universities of Waterloo and Toronto. His primary research interests within software engineering are software architecture, design tools, and software visualization. He is the author of more than 50 papers and co-author of several books, including a book recently published by Addison-Wesley titled Software Architecture in Practice. Kazman received a BA and MMath from the University of Waterloo, an MA from York University, and a PhD from Carnegie Mellon University.

Requirements and COTS-Based Systems:
A Thorny Question Indeed

David Carney

In this issue, we will focus on one of the most common words in our vocabulary, but one that has caused more than its share of trouble since people started building software systems: *requirements*. It is a powerful word. The very sound of it makes old-timers, those with 2167-A scars, roll their eyes and think of waterfalls.

Note: This column originally appeared in the June 1999 issue of SEI Interactive.

Dare we say the word?

Let it be said first off that the issue is not entirely a COTS issue; people started arguing about requirements long before COTS entered the picture. But whatever else is true, the growing use of COTS software has added fuel to the existing controversy, and it is certainly valid to speculate on how requirements play a role, given the reality of how we build systems these days. In truth, many people have already taken a very definite stand and in some quarters, use of the word "requirements" has become Politically Incorrect. (Just as in George Orwell's novel *1984*, where troublesome words are condemned to oblivion, even saying "requirements" these days can get you into trouble, especially if you're locked in a room with the Spiral/Evolutionary/Prototyping crowd.) Yet if carried to its logical end, this extreme position—don't even *think* about "requirements"—is somehow troubling: surely the person who has a need for (and who pays for!) a system to come into existence should have control over what that system is required to do.

As I consider this gnarly question, and speculate on the related issues of requirements, COTS products, and software systems, I hope the reader will bear with me as I wander down some unfamiliar paths. Though it may not be immediately obvious how a short side trip through literary history may be relevant, I promise that the excursion has some value to the topic at hand.

A visit to the dictionary

A good first step in resolving any dispute is to get agreement on what we are talking about, so it seemed logical to take this step with "requirement." The *American Heritage Dictionary* is direct and unequivocal:

> 4.   *Something that is required; necessity; something obligatory; prerequisite.*

But *Webster's*, while generally agreeing with this, adds just a bit of ambiguity:

> 5.   *Something wanted or needed.*

Whoa there. This is a little different, since this definition seems to open the door a bit from the absolute, unconditional (i.e., obligatory, requisite) into what is simply desired (i.e., wanted). So off we go to the ultimate authority, the venerable *Oxford English Dictionary*, and there some interesting lessons can be learned.

For one thing, in the *OED* the very first definition of the word "require" is "to ask," and the first several entries for the word, stretching back several centuries, are all variants of the idea of wanting; they all imply asking for something. Reaching back to Chaucer's *Canterbury Tales* (1386):

> 6.   *The firste thynge I wolde hym requere, he wolde it do.*

> 7.   *(The first thing I would* [request of him]*, he would do it.)*

To be sure, subsequent definitions (which generally reflect later historical uses of the word) gradually make the word more imperative, and begin to imply necessity, obligation, or insistence. But the initial definition of "require," maintained for several centuries, has a much stronger flavor of asking than it does of demanding.

The same is true of the initial definition of the word "requirement:" it is a request rather than a demand. The earliest use of "requirement" given by the *OED* (from 1530) is cited in precisely this sense:

> *8.   My Lord Cardinal, that obtained his legacy by our late Sovereign Lord's requirements* [e.g., petitions] *in Rome.*

The more familiar, absolutist definition of "requirement," as

> *9.   a condition which must be complied with*

is not seen until 1841, several centuries after the word entered the language:

> *10.   Has any individual…ever yet come up to their…requirements?*

So what does this have to do with software, and especially COTS software? Quite a bit, actually.

The evolving notion of software requirements

For one thing, it is fairly obvious that the term as usually used in the software engineering world refers to the later meaning of something obligatory and necessary. This was certainly its meaning in the heyday of the "waterfall" process, and at the time that the "waterfall" process was developed and popularized, it was a reasonable principle. The firm grounding provided by a collection of system requirements is the basis upon which many complex things—buildings or airplanes, for instance—are constructed, and it was perfectly normal for software engineering to share that understanding. The requirements specification was the source of all subsequent activities; in the very real sense, the requirements were in the driver's seat.

There were also reasons of historical circumstance. Most software systems were new, and therefore unprecedented. It made sense, when creating something new out of whole cloth, to specify it as rigorously as possible, through a collection of required capabilities, and then to adhere to that specification. The success of the end product was determined by a test of whether each required item of that specification had been successfully implemented.

However, notwithstanding this belief in the value of a good initial specification (and certainly long before COTS use became commonplace), there were many problems with the requirements-centric paradigm. For one thing, some systems had very long development times, and rapid advances in software technology often caught up with the system—and its requirements—and left the developing system behind. Thus, users

sometimes got "new" systems that were obsolete before they were fielded. Another problem lay in the very act of specifying the requirements, since it was often true that a system's users were not the ones who wrote the requirements. Thus, users sometimes got systems that did a lot of nice things, but not the things that they wanted or needed.

And the essential flaw in the concept was that for a complex software system, it was extremely difficult (and perhaps impossible) for a full set of system requirements to be known at all during the initial stages of development. The inevitable result was that, particularly for large systems, requirements specifications contained too much detail, and often contained inconsistent or contradictory requirements, which demanded some sort of negotiation between the competing requirements. (More on this below!)

One admirable solution to this problem lay in the notion of a "spiral" development process. First described by Barry Boehm, this concept assumes that since the full set of requirements can generally not be known when a system is begun, all attempts to "nail down" the requirements will very likely fail. To resolve the dilemma, the development process should make frequent use of prototypes, have heavy involvement with the system's end users, make ongoing refinement of the developers' understanding of the system's goals and mission, and so forth. Thus, the development process actually consists of a series of "spirals," zeroing in from a very general to a very precise knowledge of the system and what it does.

This notion of system construction is a clear improvement over the straight line "waterfall;" a spiral development process, in one form or another, is widely used on many systems today. Note, however, that an implicit idea in the spiral process is that while the requirements probably cannot be determined initially, they can eventually be determined by appropriately spiraling, prototyping, and refining. Although perhaps not known initially, the requirements for the system are somehow there, invisible perhaps, but gradually becoming more and more visible and explicit. And, presumably, still absolute.

Enter the COTS paradigm

It is not surprising that the growing use of commercial products (whether truly "off-the-shelf" or maybe needing a little tinkering) has further eroded the possibility of making an initial definition of system requirements: it is demonstrably foolish to set forth some abstract set of necessary features and then to hope that a group of COTS vendors have already created components that will just happen to meet those needs. So in a COTS-based paradigm, the concepts embedded in the spiral process—especially the idea of frequent prototyping—are especially valid and useful.

But in a more subtle manner, it may also be true that using COTS in our systems is really a semantic revolution: we may well be reverting to the centuries-old definition of

"requirement" as something we want, perhaps want very much, but that may turn out to be something we are willing to do without.

Consider how easily we are willing to use the words "requirements tradeoff." Strictly speaking, this phrase is an oxymoron: if something is a requirement (in the absolute sense), it is utterly necessary, and cannot be dispensed through any sort of negotiation, no matter what. Yet it is a familiar phrase. We used it back in the 2167-A days; that is how we decided between competing requirements. We use it even more today, and with justification. For there are many circumstances where we really do trade off our "requirements."

Suppose, for instance, that some COTS product is available and ready to hand, and we need a usable system urgently. Add to this mix a strong desire to maintain technological currency in a world that is changing with blinding rapidity. In such a case it is quite possible that we are willing to reconsider features that we thought were mandatory, reexamining them in the hard, cold light of pragmatism and practicality. Given sufficient cause (which can span a range from utility to capability to sheer economics), a "requirement" might well be abandoned; in essence, we demote it, since it is no longer a "necessity," and is by definition something that we are willing to do without.

I am not preaching anarchy here. There are still absolutes, and there must exist some set of capabilities and features that are not optional, that cannot be traded off, because without them our system will fail. But that set of items is usually smaller than the overall set of wants and desires we typically connect with our software systems. For one thing, we are now worried much more about dollars: part of the new acquisition reform initiative is the notion of "cost as an independent variable" (CAIV), premised on this very principle of requirements negotiation. According to CAIV, requirements tradeoff against cost should be explicitly considered in the design process; those "requirements" that escalate costs are to be revised, a far cry from the days when cost was a non-requirement.

Distinguishing the real absolutes from all the rest is the key. We need better self-knowledge about the bottom line of what we demand from our systems, and we must distinguish that bottom line from all of the other system goals. Like the familiar movie scenario of the gullible tourist entering the bazaar in the Casbah, we need to know clearly what we want, what we will settle for, and what we are willing to pay. Otherwise, the movie scenario has a *very* unhappy ending.

In short, our choice of using COTS products in our systems expands and cements the notion that requirements need to share the driver's seat. Since we choose to accrue the many benefits that come from letting the vendors develop COTS products, we must therefore be willing to permit some parts of our systems—the "requirements" that drove the creation of those COTS pieces—to be controlled by those same vendors. And the law

of transitivity holds here as well: if vendors have control over their own products, then some degree of the control over our fielded system is therefore in their hands, not in ours.

Last thoughts

In a rather contorted way, I guess I am claiming that the way the word "requirement," as used in COTS system building and especially in the notion of "requirements tradeoff," is historically justified. By viewing "requirements" as a set of wants and desires, all of which reside on a sliding scale from absolute down through desirable to simple "nice-to-haves," we are, in reality, returning to the original meaning of the word. (And perhaps we should think that those who use the phrase "requirements tradeoff" are actually linguistic Puritans at heart, and closet Chaucer scholars to boot.)

Maybe so, maybe not. But it is certainly true that for a COTS-based system, the more flexible meaning of "requirement" is the only one that makes sense. As we specify our systems today, we create some collection of system features that is really a bunch of things we want, but that includes a lot of things that we know we won't get. And the act of "trading off" involves negotiation, giving and taking, weighing the risks we take if we give up certain capabilities, and similar unfamiliar but critical activities.

One last thought: During this brief consideration of how requirements figure in COTS-based systems, one thing has been a subtle and constant presence, and warrants further discussion. That is, in pursuing a COTS-based acquisition strategy and a COTS-based development paradigm, almost all of the new or novel things that we now must do (requirements tradeoffs being just one of them) somehow revolve around our loss of control over certain aspects of our systems. This loss of control is manifest in functionality, in schedule, in dependencies, and in other less obvious ways. It is a very interesting and vexing issue, and will be the topic of my next column. Stay tuned.

About the author

David Carney is a member of the technical staff in the Dynamic Systems Program at the SEI. Before coming to the SEI, he was on the staff of the Institute for Defense Analysis in Alexandria, Va., where he worked with the Software Technology for Adaptable, Reliable Systems program and with the NATO Special Working Group on Ada Programming Support Environment. Before that, he was employed at Intermetrics, Inc., where he worked on the Ada Integrated Environment project.

How the FBI Investigates Computer Crime

DKS with Eric Hayes, CERT/CC

Imagine your surprise if one ordinary day at work you receive an email claiming that your company's computers were used to help launch a major denial-of service attack, or if you receive a call from management saying that someone is threatening to expose corporate trade secrets unless they receive a big payoff? Or imagine your dismay if you discover a fellow employee has used your company's computer to illegally trade Metallica songs! What do you do?

You're a Victim; Now What?

For many companies today, being the victim of computer crime, whether it is simple misuse or a major violation, is no longer a rare occurrence. What happens next? Trying to discover and repair the damage is just part of the story. For many people responsible for network and computer security, the next step is to take a deep breath, reach for the phone and call the Federal Bureau of Investigation (FBI). This article (originally published in collaboration with the FBI as a CERT® Coordination Center [CERT/CC] tech tip [http://www.cert.org/tech_tips/FBI_investigates_crime.html]) explains some of the guidelines, policies and resources used by the FBI when it investigates computer crime and gives you some ideas about how you can help an investigation succeed.

The FBI has implemented various technical programs to address the growing complexity of computer investigations. FBI legal attachés stationed in 41 countries enable the FBI to use sophisticated methods to investigate and coordinate a response to cyber incidents around the world. In Washington, DC, the National Infrastructure Protection Center (NIPC) is a special unit that coordinates computer crimes investigations throughout the United States. The FBI trains and certifies computer forensic examiners for each of the 56 FBI field offices in the United States to recover and preserve digital evidence. The FBI maintains a computer forensic laboratory in Washington, DC, for advanced data recovery and for research and development.

Computer crimes can be separated into two categories: (1) crimes facilitated by a computer and (2) crimes where the computer is the target (the focus of this article). Computer-facilitated crime occurs when a computer is used as a tool to aid criminal activity. This can include storing records of fraud, producing false identification, reproducing and distributing copyright material, collecting and distributing child pornography, and many other crimes.

Crimes where computers are the targets are unlike traditional types of crimes. Technology has made it more difficult to answer the questions of who, what, where, when, and how. Therefore, in an electronic or digital environment, evidence is now collected and handled differently from how it was handled in the past.

Federal Statutes Used in Computer Crimes

The FBI uses a number of federal statutes to investigate computer crimes. The following are used most frequently:

- 18 United States Code (U.S.C.) 875 Interstate Communications: Including Threats, Kidnapping, Ransom, Extortion
- 18 U.S.C. 1029 Possession of Access Devices
- 18 U.S.C. 1030 Fraud and Related Activity in Connection with Computers
- 18 U.S.C. 1343 Fraud by Wire, Radio or Television
- 18 U.S.C. 1361 Injury to Government Property
- 18 U.S.C. 1362 Government Communication Systems
- 18 U.S.C. 1831 Economic Espionage Act
- 18 U.S.C. 1832 Trade Secrets Act

For more information see: http://www.usdoj.gov/criminal/cybercrime/fedcode.htm.

Note: Each state has different laws and procedures that pertain to the investigation and prosecution of computer crimes. Contact your local police department or district attorney's office for guidance.

The FBI is sensitive to a victim's concerns about public exposure, so any decision to investigate is jointly made between the FBI and the United States Attorney and takes the victim's needs into account.

The FBI investigates incidents when both of the following conditions are present:

- a violation of the federal criminal code has occurred within the jurisdiction of the FBI
- the United States Attorney's Office supports the investigation and agrees to prosecute the subject if the elements of the federal violation can be substantiated

Federal law enforcement can only gather proprietary information concerning an incident in the following ways:

- request for voluntary disclosure of information

- court order

- federal grand jury subpoena

- search warrant

The following steps will help you document an incident and assist federal, state, and local law enforcement agencies in their investigations (be sure to act in accordance with your organization's polices and procedures):

1. Make sure that staff members know who in the organization is responsible for cyber security and how to reach them.

2. Preserve the state of the computer at the time of the incident by making a backup copy of logs, damaged or altered files, and files left by the intruder.

3. If the incident is in progress, activate auditing software and consider implementing a keystroke monitoring program. (Make sure the system log-on warning banner permits a monitoring program to be implemented.)

4. If you have reported the incident to the CERT/CC, consider authorizing it to release the incident information to law enforcement. This will provide an excellent synopsis of what happened. The CERT/CC Incident Report Form is located at http://www.cert.org/reporting/incident_form.txt.

5. Document all losses your organization suffered as a result of the incident. These could include

   a. the estimated number of hours spent in response and recovery (multiply the number of participating staff by their hourly rates)

   b. the cost of temporary help

   c. the cost of damaged equipment

   d. the value of data lost

   e. the amount of credit given to customers because of the inconvenience

   f. the loss of revenue

   g. the value of "trade secret" information

6. Contact law enforcement and

   a. provide incident documentation

   a. share information about the intruder

   b. share your ideas about possible motives

To initiate an investigation, contact your local FBI office or the appropriate federal, state, or local law enforcement agency. To report an incident, call the FBI NIPC Watch and Warning Unit at (202) 323-3205.

For further information, see the NIPC home page: http://www.nipc.gov

About the Authors

DKS is a special agent with the Federal Bureau of Investigation.

Eric Hayes is a member of the technical staff and a senior technical writer/editor in the Networked Systems Survivability (NSS) Program at the Software Engineering Institute (SEI). The CERT Coordination Center is a part of this program. Before joining the SEI, Hayes worked in the Information Services Department at the Norwest Corporation as an editor of standard operating procedures (SOP) manuals and served as the team lead for SOP editors. Prior to that, he founded Hayes Communications, which offered services such as marketing, fundraising, research, Web page production, and public relations writing. Hayes received a BA in English writing from the University of Pittsburgh. At the graduate level, he has studied rhetoric at the University of Wisconsin at Milwaukee, technical editing at the University of Minnesota at Minneapolis, and technical writing at Carnegie Mellon University. Hayes is a member of the Society for Technical Communication.

The Future of Software Engineering: I

Watts S. Humphrey

In this and the next few columns, I discuss the future of software engineering. This column focuses on trends in application programming, particularly as they concern quality. In subsequent columns, I address programming skills, trends in systems programming, and the implications of these trends for software engineering in general. While the positions I take and the opinions I express are likely to be controversial, my intent is to stir up some debate and hopefully to shed light on what I believe are important issues. Also, as is true in all of these columns, the opinions I express are entirely my own.

Current Trends

Some 50 years ago when I finished graduate school, I started to work with computers. For almost all of the intervening time, people have been dreaming up new and better ways to use computing devices and systems. While the volume of application development has grown steadily, so has the list of applications that people want to develop. It seems that the more programs we write, the more we understand what we need, and the more programs we add to the application-development backlog. So far, the rate of growth in application demand has continued to accelerate.

As economists say, unsustainable trends are unsustainable. However, to date, there is no sign that this growth in demand is slowing. I suspect that it will continue for the foreseeable future, though we will certainly see changes in the types of application programs. The reason for this growth is that application programming is a very effective way to meet many kinds of human needs. As long as people continue devising new and cleverer ways to work and to play, we can expect the growth in computer applications to continue. The more we learn about computing systems and the more problems we solve, the better we understand how to address more complex and interesting problems. Therefore, because human ingenuity appears to be unlimited, the number of programs needed in the world is also essentially unlimited. This means that the principal limitation on the expanding use of computing systems is our ability to find enough skilled people to meet the demands.

In discussing these topics, I break the software world into two broad categories: applications and systems. I won't try to clearly define the dividing line between these categories because that line is both indistinct and changing. By "application programming," I refer to solving human problems with computing systems. Here the focus is on defining the problem and figuring out how to solve it with an automated system. Systems programming focuses on how to provide automated systems, aids, tools,

and facilities to help people produce and run application programs. In looking toward the future in these areas, I believe that the two most significant issues concern software quality and the demand for skilled people. I address application software quality first.

Application Program Quality

The quality of application software today is spotty at best. A few programs are of high quality and many are downright bad. The discussion of application quality has two principal parts. First, computers are being used for progressively more critical business applications. This means that, even at current quality levels, the impact of software defects will grow. This implies that the demand for high-quality application software will also grow.

To date, people buy and use software without apparent concern for their quality. While they complain about quality problems, software quality has not yet become a significant acquisition consideration. Until it is, we cannot expect suppliers to measure and manage the quality of their products. However, as has been amply demonstrated in other fields, when quality is not measured and managed, it is generally poor.

When the public gets concerned about quality, these attitudes will quickly change. It will not take many high-profile disasters to cause executives to worry. Then, they are likely to demand quality guarantees and warranties from their suppliers before they entrust their businesses to new computing systems. When customers cannot be assured of the quality of a software package, they will either go to a competitor or forgo the application entirely and continue using prior methods. This would not be because automated methods would not have been helpful, but simply because the risk of failure would be too high.

Growing Program Complexity

The second part of the program quality discussion concerns size and complexity. The size and complexity of application programs is increasing. The size data in Figure 1 show just how fast this growth has been. The IBM size measures are from my personal recollection, the Microsoft NT size data are from published reports, and the spacecraft size data are from Barry Boehm [Boehm 81, Zachary 94]. The TV data are for the embedded code in television sets and are from a talk by Hans Aerts and others at the European SEPG conference in June 2000. According to my prior definitions, the IBM and Microsoft products are system software while the spacecraft and TV programs are application software.

These data show that the size of the software required for various kinds of systems and applications has been growing exponentially for the past 40 years. The trend line in the center of the chart shows a compound growth rate of ten times every five years. This is the same as Moore's law for the growth in the number of semiconductors on a chip, or a doubling every 18 months.

While the size growth of system software has been phenomenal, it appears to be slowing, at least from the appearance of the IBM and NT data in Figure 1. I discuss these systems software trends in a later column. The critical point from an application quality point of view is that the growth trend for applications software appears to be continuing.

The Defect Content of Programs

Assuming that the same methods are used, the number of defects in a program increases linearly with its size. However, the rate at which application program users experience problems is largely determined by the number of defects in a program rather than their density. Therefore, even though the defect density may stay about the same or even improve, merely producing larger programs with the same methods will produce progressively less reliable systems. So, either the quality of future application software must improve—at least in step with the increasing sensitivity of the applications—or businesses must limit their use of computing systems to less critical applications.
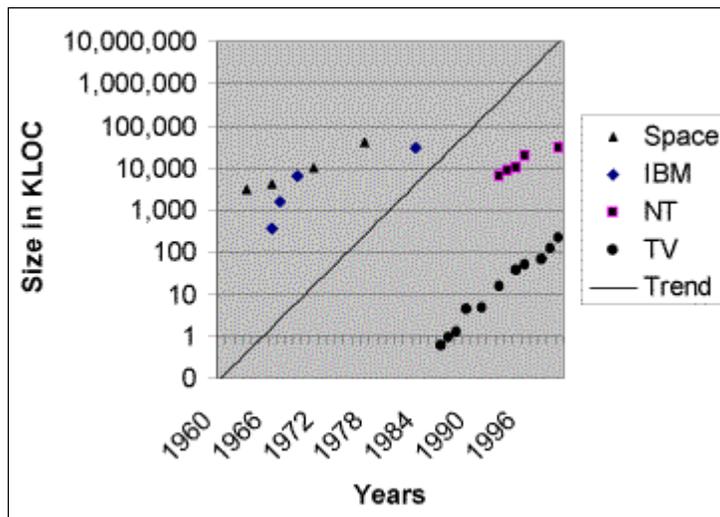

Figure 1: Program Size Growth

To see why program reliability depends on defect numbers instead of defect density, consider an example. A 200 KLOC (thousand lines of code) program with 5 undetected defects per KLOC would have 1,000 defects. If you replaced this program with an

enhanced program with 2,000 KLOC and the same defect density, it would have 10,000 defects. Assuming that the users followed a similar usage cycle with the new application, they would exercise the larger program at about the same rate as the previous one. While this would presumably require a faster computer, the users would be exposed to ten times as many defects in the same amount of time. This, of course, assumes that the users actually used many of the new program's enhanced functions. If they did not, they presumably would not have needed the new program.

An Application Quality Example

Oil exploration companies use highly sophisticated programs to analyze seismic data. These programs all use the same mathematical methods and should give identical results when run with identical data. While the programs are proprietary to each exploration company, the proprietary parts of these programs concern how they process enormous volumes of data. This is important because the volume of seismic data to be analyzed is often in the terabyte range.

A few years ago, Les Hatton persuaded several oil-exploration companies to give him copies of nine such programs [Hatton 94]. He also obtained a seismic exploration dataset and ran each of these nine programs with the identical data. The results are shown in Figure 2. Here, the range of calculated values is shown for several consecutive program iterations. As you can see, this range generally increased with the number of cycles, and after a few runs, it reached 100%. When one of these companies was told about some of the conditions under which its program gave unusual results, the programmers found and corrected the mistakes, and the program's next results agreed with the other programs.
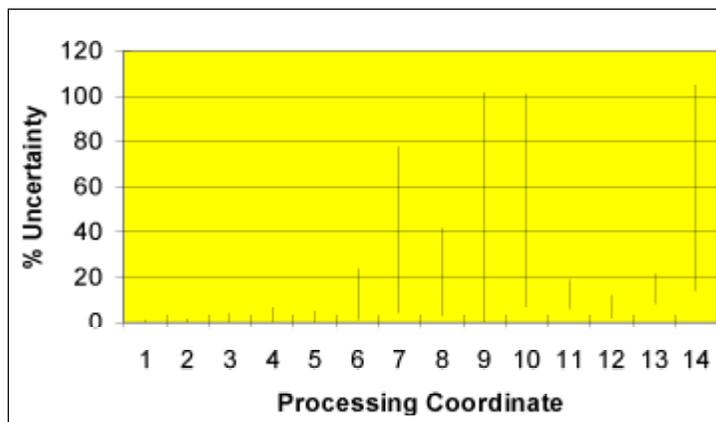

Figure 2: Growth in Seismic Program Uncertainty

The results produced by these oil-exploration programs contained errors of up to 100%, and these results were used to make multi-million-dollar decisions on where to drill oil wells. Based on this study, it appears that these programs provided little better guidance

than throwing dice. I am not picking on these programs as particularly poor examples. Their quality appears to be typical of many application programs.

The Quality Problem

In discussing the quality of application programs, we need to consider the fact that defective programs run. That is, when engineers produce a program and then run extensive tests, they generally can get it to work. Unfortunately, unless the tests were comprehensive, the tested program will likely contain a great many defects.

Any testing process can only identify and fix the defects encountered in running those specific tests. This is because many program defects are sensitive to the program's state, the data values used, the system configuration, and the operating conditions. Because the number of possible combinations of these conditions is very large, even for relatively simple programs, extensive testing cannot find all of the defects.

Since the size of application programs will continue to increase, we need to consider another question: will we be able to test these programs? That is, how well does the testing process scale up with program size? Unfortunately, the answer is not encouraging. As programs get larger, the number of possible program conditions increases exponentially. This has two related consequences.

1. The number of tests required to achieve any given level of test coverage increases exponentially with program size.
1. The time it takes to find and fix each program defect increases somewhere between linearly and exponentially with program size.

The inescapable conclusion is that the testing process will not scale up with program size. Since the quality of today's programs is marginal and the demand for quality is increasing, current software quality practices will not be adequate in the future.

The Impact of Poor Quality

As the cost of application mistakes grows and as these mistakes increasingly impact business performance, application program quality will become progressively more important. While this will come as a shock to many in the software community, it will be a positive development. The reason is that suppliers will not generally pay attention to quality until their customers start to demand it. When software quality becomes an important economic consideration for businesses, we can expect software organizations to give it much higher priority.

While one could hope that the software industry would recognize the benefits of quality work before they are forced to, the signs are not encouraging. However, improved product quality would be in the industry's best interests. It would mean increased opportunities for computing systems and increased demand for the suppliers' products. This would also mean continued growth in the demand for skilled software professionals.

In the next column, I discuss the need for application programming skills and how this need is directly related to the quality problem. Following that, I discuss related trends in systems programming and the implications of these trends for software engineering.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Sholom Cohen, Noopur Davis, Alan Koch, Don McAndrews, Julia Mullaney, Bill Peterson, and Marsha Pomeroy-Huff.

In closing, an invitation to readers

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. However, I am most interested in addressing issues that you feel are important. So, please drop me a note at watts@sei.cmu.edu with your comments, questions, or suggestions. I will read your notes and consider them in planning future columns.

Thanks for your attention and please stay tuned in.

References

[Boehm 81]   Boehm, Barry. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[Hatton 94]   Hatton, Les. "How Accurate is Scientific Software?" *IEEE Transactions on Software Engineering*, Vol. 20, No. 10 (October 1994): 785-797.

[Zachary 94]   Zachary, G. Pascal. *Showstopper!* New York: The Free Press, 1994.

About the Author

Watts S. Humphrey founded the Software Process Program at the SEI. He is a fellow of the institute and is a research scientist on its staff. From 1959 to 1986, he was associated with IBM Corporation, where he was director of programming quality and process. His publications include many technical papers and six books. His most recent books are *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software Process*^{SM}(1997). He holds five U.S. patents. He is a member of the Association for Computing Machinery, a fellow of the Institute for Electrical and Electronics Engineers, and a past member of the Malcolm Baldrige National Quality Award Board of Examiners. He holds a BS in physics from the University of Chicago, an MS in physics from the Illinois Institute of Technology, and an MBA from the University of Chicago.