The Double-Edged COTS IT Sword

Lt. Col. Joe Jarzombek
U.S. Air Force ESIP Director

As government technical teams downsize and budgets shrink in tandem with an increasing demand for more complex systems, there is a rising interest in leveraging the use of commercial-off-the-shelf (COTS) products. In many cases, the use of COTS is mandated. Is it possible to over-emphasize the use of COTS products? For example, what checks would you expect prior to flying on a new aircraft with a software system composed of integrated COTS? Far from the promised panacea, the use of COTS components introduces new trade-offs and issues, especially with risk management, component integration, system reliability, and cost of sustainment.

Is there a limit to what can be defined as COTS? Can all future deliverables of a developed product be considered COTS? If so, what is the implication for Department of Defense (DoD) acquisitions? The new DoD Directive 8000.1 for the Management of DoD Information and Information Technology (IT) (http://www.cio.hq.af.mil/dodctext.htm) sharpens the blades of COTS and outsourcing policies by providing the mechanism for DoD compliance with the Cohen-Clinger Act (also known as the IT Management Reform Act of 1996 (see *CROSSTALK*, September 1997). It applies to all DoD IT, even IT in national security systems (to include embedded, crypto, intelligence, and command and control systems). In other words, the law and the new DoD directive provide an all-encompassing definition of information and IT. DoD Directive 8000.1 requires an assessment of where the IT function could be performed most effectively: within DoD, by another government source, or in the private sector. It also advocates the principal of fee-for-service in governing the provisioning of information services and IT capabilities. Considering the law and the new DoD directive, how far can the use of COTS be applied (and pressure applied by external sources)? Consider space-based capabilities that have been delivered before, especially since they are now covered by the IT umbrella. Why couldn't DoD simply be expected to request an on-orbit COTS (or COTS-based) system with a specified capability, capacity, and availability in a particular orbit by a specified date? Based on the law and the DoD IT directive, this must be considered a viable alternative.

COTS software offers the ability to quickly adapt to evolving mission and business environments with lower up-front costs; however, many projects are finding that the promise of COTS components does not quite match expectations. The May 1997 issue of *CROSSTALK* offered several articles that tackled the relevant question, "Is COTS worthy of worship?" "The Commandments of COTS: Still in Search of the Promised Land" is worth reviewing and should be provided to senior leaders who believe COTS is the ultimate answer to all their software challenges.

In this issue, "The Opportunities and Complexities of Applying Commercial-Off-the-Shelf Components" (page 4) provides managers with a better understanding of COTS. The COTS mandate challenges system developers to integrate COTS components into systems without compromising the strict reliability and availability required in mission-critical systems. Most COTS components are essentially "black boxes" with no warranty. Systems must maintain their exist-

see COTS, page 3

Call for Articles

If your experience or research has produced information that could be useful to others, *CROSSTALK* will get the word out. Not only is *CROSSTALK* a forum for high-profile leaders, it is an effective medium for useful information from all levels within the Department of Defense (DoD), industry, and academia.

Published monthly, *CROSSTALK* is an official DoD periodical distributed to over 19,000 readers, plus uncounted others who are exposed to the journal in offices, libraries, the Internet, and other venues. *CROSSTALK* articles are also regularly reprinted in other publications.

We welcome articles on all software-related topics, but are especially interested in several high-interest areas. Drawing from reader survey data, we will highlight your most requested article topics as themes for 1998 *CROSSTALK* issues. In future issues, we will place a special, yet nonexclusive, focus on

Process Improvement – *September 1998*
Article Submission Deadline: May 4, 1998

Systems Engineering – *October 1998*
Article Submission Deadline: June 1, 1998

Software Quality – *November 1998*
Article Submission Deadline: July 6, 1998

Look for additional announcements that reveal more of our future issues' themes. We will accept article submissions on all software-related topics at any time; our issues will not focus exclusively on the featured theme.

Please follow the *Guidelines for CROSSTALK Authors,* available on the World Wide Web at http://www.stsc.hill.af.mil/. Hard copies of the guidelines are also available upon request. All articles must be approved by the *CROSSTALK* Editorial Board prior to publication. We do not pay for articles. Send articles to

Ogden ALC/TISE
ATTN: Heather Winward, *CROSSTALK* Features Coordinator
7278 Fourth Street
Hill AFB, UT 84056-5205

Or E-mail articles to winwardh@software.hill.af.mil. For additional information, call 801-777-9239.

Forrest Brown
Managing Editor

No. 228-M
MEMORANDUM FOR CORRESPONDENTS December 16, 1997

The Office of the Under Secretary of Defense for Acquisition and Technology (OUSD(A&T)) recently recommended that Capability Maturity Model (CMM™) Integration (CMMI) be the number one Software Engineering Institute's (SEI) priority in its process management work. The Software Engineering Institute is located at Carnegie Mellon University in Pittsburgh, Pa. The department has recommended that SEI construct tailored CMMs from both common building blocks and discipline-specific elements. This approach will ensure consistency among all tailored CMMs and will eliminate the need to apply a variety of CMMs to a single organization.

"The Department of Defense vision is that existing and future CMMs will be integrated into one framework which addresses Acquisition Reform, process improvement from an integrated product and process development perspective and contain sound principles of systems development," said Mark Schaeffer, deputy director for systems test, systems engineering and evaluation, OUSD(A&T).

The new approach for CMM integration is different from that of the SEI's first release of the Common CMM Framework. The SEI is pursuing this new strategic approach to meet its sponsor's and customers' needs for integrated CMMs that range from single domain to enterprise-wide within the same framework. The SEI developed the Software Capability Maturity Model (SW-CMM), which has been widely adopted and used throughout both the government and corporate worlds. The SEI's sponsor requested the SEI to delay the release of SW-CMM, Version 2.0 for several months until the CMMI Framework can be defined and approved. This is also intended to reduce confusion among customers as a process improvement framework for multiple disciplines is developed.

Meanwhile, the SEI continues to support SW-CMM, Version 1.1 and its associated products. The information contained in Draft C of SW-CMM, Version 2.0, which is currently available on the SEI Web site, will provide industry partners advanced notice of the changes to Levels IV and V processes, and allow comments to be fed back for incorporation into CMMI. The Department of Defense (DoD) and the SEI encourage all involved in software process improvement to review and provide feedback on Draft C of the SW-CMM, Version 2.0 publication, which is on the SEI Web site at the URL: http://www.sei.cmu.edu/technology/cmm/cg.html.

For further information concerning CMMI, please contact Terrence McGillen, Software Engineering Institute, Carnegie Mellon University, 412-268-7394. The DoD Public Affairs point of contact is Lt. Col. Bob Potter, 703-697-3189.

COTS, from page 2

ing level of performance even when upgraded components are introduced.

A software fault-tolerant architecture is needed to help developers modify existing applications with upgraded COTS software components. The Software Engineering Institute has developed a framework called the "Simplex Architecture" that addresses the challenges of using COTS in high-reliability systems (page 7). The framework integrates high-assurance application-kernel technology, address-space protection mechanisms, real-time scheduling algorithms, and methods for dynamic communication among modules.

Although COTS products offer challenges, COTS software is a viable means to cost-effectively satisfy mission requirements. To minimize the need to develop unique systems, I continue to advise project teams to consider COTS when defining operational requirements and business processes. Some people advocate disregarding existing products and services when defining requirements and processes so they will not be bound by existing technology, but this can lead to reinventing the wheel. Projects need to have people who are knowledgeable of the plethora of existing COTS products and services so they can recommend tailoring opportunities when defining requirements and processes. If you do not have at least one COTS-knowledgeable person on your team who can articulate the project realities of using COTS products, you should seek that support.

I suggest you read the new DoD Directive 8000.1 to understand how it applies to your projects. In light of COTS and outsourcing mandates and the new DoD IT directive, project managers need to understand what lawmakers, audit agencies, and senior DoD leaders might be considering in reviewing their programs. ◆

The Opportunities and Complexities of Applying Commercial-Off-the-Shelf Components

Lisa Brownsword, David Carney, and Tricia Oberndorf
Software Engineering Institute

Government acquisition policies for software-intensive systems emphasize the use of commercial-off-the-shelf (COTS) products. On the surface, "the COTS solution" appears straightforward. In actuality, many projects find its use less than straightforward. This article provides acquisition managers and policy makers with a basic understanding of how developing systems with COTS products is different and why and what new COTS capabilities are being identified.

Government acquisition policies for software-intensive systems now emphasize the use of existing commercial products. Requests for Proposals often require the use of specific COTS products and sometimes specify the amounts to be used. As systems are reengineered, many include the use of COTS products. And as government budgets shrink and the desire for increasingly complex systems continues, there is rising interest in leveraging the use of commercially available products when possible.

Although on the surface "the COTS solution" appears straightforward and compelling, projects that apply COTS find its use less than straightforward. Rather, they encounter significant new trade-offs and issues. Applying COTS products is not merely a technical matter for system integrators. It has a profound impact on business, acquisition, and management practices, and organizational structures. Compounding the problem is the limited experience and guidance currently available on how to effectively approach system development with commercial components.

The Software Engineering Institute (SEI), along with other key organizations associated with the government and civil agencies, is creating and assembling best practice guidance for acquisition and program managers, integrators, and testers through case studies, hands-on support, and analysis. This article is one of several venues the SEI is leveraging to provide acquisition managers and policy makers with an understanding of what is different in the development of systems with COTS products and why and what new capabilities are being identified. Due to the brevity of this article, the discussion is limited to a few essential aspects of the differences and capabilities of COTS products.

What Is Different with COTS-Based Systems?

COTS products can be applied to a spectrum of systems. At one end of the spectrum are nearly packaged software solutions, such as Microsoft Office or Common Desktop Environment, that require no integration with other components. These systems map well to the needs and operations of the government.

Further along the spectrum are COTS products that support the information management domain, such as Oracle or Sybase. These systems typically consist of both COTS products and custom components, with COTS products making up the majority of the system. Depending on how well the COTS products and custom components fit together, a small to moderate amount of customization is usually required to enable them to work cooperatively.

At the other end of the spectrum, there are systems composed of a complex mix of commercial and noncommercial products that provide large-scale functionality that is otherwise not available. Such systems typically require large amounts of "glue" code to integrate the set of components. These systems are typically in the embedded, real-time, or safety-critical domains.

Using COTS products for applications at the packaged software solutions end of the spectrum is relatively straightforward; however, using them for complex systems further along the spectrum is not. This article focuses on the issues and complications that arise when constructing complex systems with COTS products.

Fundamental Paradigm Change

Traditionally, organizations develop systems from scratch with control over all or most of the pieces. They

- collect and define requirements.
- identify an architecture to satisfy the requirements.
- design in detail individual subsystems to fit within the architecture.
- code, test, and debug modules to meet the specified requirements.
- integrate sets of modules and subsystems into the complete system.

With the use of COTS as components for a system, a fundamental change occurs: an organization now *composes* the system from building blocks that may or may not (generally do not) work cooperatively directly out of the box. The organization will require skilled engineering expertise to determine how to make a set of components work cooperatively—and at what cost.

This fundamental shift from development to composition causes numerous technical, organizational, management, and business changes. Some of these changes are obvious, whereas others are quite subtle, but if not addressed, either type of change can cause severe problems for the project. Consequently, organizations may have to modify their

procedures and structures and, in some cases, create entirely new procedures.

Impact on Typical Lifecycle Activities

Regardless of which lifecycle model an organization uses (waterfall, spiral, or iterative), they perform requirements, architecture, detailed design, code, test, and system integration activities. The use of COTS products has a pervasive impact on all lifecycle activities. This is illustrated by briefly examining the impact to requirements, testing, and maintenance activities.

Requirements describe the desired system behavior and capability with a set of specified conditions. For a COTS-based system, the specified requirements must be sufficiently flexible to accommodate a variety of available commercial products and their associated fluctuations over time. To write such requirements, the author must know enough about the commercial marketplace to describe functional features for which commercial products exist.

There is a critical relationship among technology and product selection, requirement specification, and architecture definition. If you define your architecture to fulfill your requirements and then select your COTS products, you may have only a few or no available products that fit within the chosen architecture. Pragmatically, three essential elements (requirements, architecture, and product selection) must be worked in parallel with constant trade-offs among them.

As the testing of COTS-based systems is considered, you must determine what levels of testing are possible or needed. A COTS product is a "black box" and therefore changes the nature of testing. A system may use only a partial set of features of a given COTS product. Should you test only the features used in the system? How do you test for failures in used features that may have abnormal behavior due to unknown dependencies between the used and unused features of a COTS product?

Maintenance also changes in fundamental ways—it is no longer solely concerned with fixing existing behavior or incorporating new mission needs. Vendors update their COTS products on their own schedules and at differing intervals. Also, a vendor may elect to eliminate, change, add, or combine features for a release. Updates to one COTS product, such as new file formats or naming convention changes, can have unforeseen consequences for other COTS products in the system. To further complicate maintenance, all COTS products require continual attention to license expirations and changes. All these events routinely occur. All these activities may (and typically do) start well *before* an organization fields the system or major upgrade. Pragmatically, the distinction between development and maintenance all but disappears.

Emergence of Nontypical Activities

We can view the commercial marketplace as a continuous "product conveyor belt"—the marketplace constantly adds new products and technology to the belt, existing products evolve through continuous upgrades, and vendors remove products from the marketplace. The government has limited influence (and no direct control) over the speed, content, or variety of products on the product belt.

Consumers, such as the government, must constantly keep abreast of the state of the product belt. This requires new activities (with associated resource requirements) in the area of technology and product evaluation. Consumers must identify potential technologies and products, qualify candidates for fit within their system, and perform trade-off analysis between competing technologies and products. An organization must *continuously* perform the entire process of monitoring, evaluating, qualifying, and analyzing the impact of technology and products given the constant changes within the commercial marketplace. We should add that technology and product awareness and evaluation are not activities that the government can merely relegate to its contractors. The government must also have such a capability if it is to specify and manage its systems wisely.

Assembling COTS products also presents new difficulties. Although software COTS products are attempting to simulate the "plug-and-play" capability of the hardware world, in today's reality, software COTS products seldom plug into anything easily. Most products require some amount of adaptation to work harmoniously with the other commercial or custom components in the system. The typical solution is to adapt each software COTS product through the use of "wrappers," "bridges," or other "glueware." It is important to note that adaptation does *not* imply modification of the COTS product. However, adaptation can be a complex activity that requires technical expertise at the detailed system and specific COTS component levels. Adaptation must take into account the interactions among custom components, COTS products, any nondevelopmental item components, any legacy code, and the architecture, including infrastructure and middleware elements. This adaptation process has a cost—a potentially high one.

What Should an Acquisition or Program Manager Do to Get Started Using COTS?

Applying a COTS solution requires the government to create and maintain new competencies. We have alluded to a number of essential capabilities throughout the article. The following sections identify a number of actions to establish an effective infrastructure for the use of COTS products. These actions are not intended as a road map or to be all-inclusive. Rather, they are a set of practical actions to help organizations *start* to develop the necessary knowledge and experience base. We recommend that organizations begin now—ideally before the first (or next) COTS-based system development, reengineering, or maintenance project.

Know the Regulations

There are various policies, regulations, and directives relative to the general use of commercial products. Policies also exist concerning the use of specific COTS products such as the Distributed Information Infrastructure Com-

mon Operating Environment. Understand what policies and directives apply and how they apply to your particular systems. Situations or directives may change. Therefore, an organization should have a "regulations guru" available whose ongoing work is to remain informed of the various government regulations and their impact to the organization's systems.

Know Your Marketplace

The COTS marketplace is huge and continually changes. Determine which subsets of the marketplace are relevant to your systems. Develop dedicated resources to become conversant with the available and emerging COTS technologies and products, and determine their impact for your applications.

Know How to Evaluate Technologies and Products

Determining which products and technologies are most appropriate for a given system requires more than a market survey based on marketing literature and vendor demonstrations. Know how to develop evaluation criteria, conduct a satisfactory evaluation, and select viable technologies and products based on your criteria. Determine the amount of time to allocate for evaluation during the acquisition. Leverage previous projects to experiment with developing the expertise required.

Know How to Develop Requirements

It is vital to understand how to specify requirements; this strikes the optimum balance between desired user functionality and the available COTS product. Know how to make trade-offs between COTS products, your architecture, and your requirements. Again, leverage previous projects to experiment with developing the required expertise.

Know How to Manage System and COTS Product Evolution

Learn how the development and maintenance of your systems will need to change as a result of the continual release of COTS product updates. Some sample areas to investigate include scheduling of COTS product updates into your baseline, impact analysis to determine potential interactions and changes to existing components, impact analysis to the operations of the fielded system, and identification and management of licensing issues for your intended COTS products.

Determine How to Build Your Business Case

Although the motivation for the use of COTS products for many organizations is cost savings, an organization should address the many business unknowns prior to making that determination. How are the costs of both the initial and recurring adaptation throughout maintenance determined? How should a program manager make the business case if the total lifecycle cost is higher?

Develop a Metrics Database to Determine Your Business Case

Currently, there is little data on the cost, schedule, or quality benefits of COTS-based systems. Begin collecting the data needed to develop a realistic business case. Such data might include cost, time distribution across lifecycle activities, defects after the system is fielded, or efficiencies gained or lost in field operations.

Final Remarks

Even if an organization obtains some parts from commercial sources, a COTS-based system is still a system with requirements. Only the people who pay for, maintain, and use a COTS-based system are concerned about the quality of the system—vendors are not. Organizations must still design, assemble, test, manage, and maintain the system. There is no magic. There is no COTS "silver bullet." The government's responsibility for its systems is not eliminated or reduced by a reliance on COTS products.

COTS systems require acquisition and program managers, policy makers, systems integrators, and system designers to become smart consumers by understanding the business, management, organizational, and technical implications of applying COTS products to system development or reengineering. The worst thing you can do is treat the shift toward the use of COTS products as merely a change in technology. ◆

About the Authors

Lisa Brownsword is a member of the technical staff in the Dynamic Systems program at the SEI. Before the SEI, she worked for Computer Sciences Corporation in support of the NASA/Goddard Software Engineering Laboratory. Prior to that, at Rational Software Corporation, she provided consulting to managers and technical practitioners in the use of software engineering practices, including architecture-centered development, product lines, object technology, Ada, and computer-aided software engineering (CASE).

Software Engineering Institute
Carnegie Mellon University
4301 Wilson Boulevard, Suite 902
Arlington, VA 22203
Voice: 703-908-8203
Fax: 703-908-9317
E-mail: llb@sei.cmu.edu

David Carney is a member of the technical staff in the Dynamic Systems program at the SEI. Before joining 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 the Ada Programming Support Environment. Prior to that, he was employed at Intermetrics, Inc., where he worked on the Ada Integrated Environment project.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 412-268-6525
Fax: 412-268-5758
E-mail: djc@sei.cmu.edu

Tricia Oberndorf is a member of the technical staff at the SEI. She is a part of the Dynamic Systems program and concentrates on the investigation of integration and open systems issues. She has investigated a number of other integration and open systems questions in the context of CASE environments and other kinds of systems. Prior to the SEI, she worked for the Navy for more than 19 years.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 412-268-6138
Fax: 412-268-5758
E-mail: po@sei.cmu.edu

Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems

Lui Sha, John B. Goodenough, and Bill Pollak
Software Engineering Institute

Since the end of the Cold War and the downsizing of military budgets, it has been more important than ever that mission-critical systems be reliable, affordable, and capable of evolving to prevent obsolescence. Furthermore, as operational software systems play a more critical role in both military and nonmilitary applications, the need for dependability in all software systems is increasing. In this article, we review a new combination of existing technologies that can meet these challenges.

The Challenges

To cut costs and gain leverage from technical advances in the commercial sector, the Department of Defense (DoD) has actively encouraged the more frequent use of commercial-off-the-shelf (COTS) components in its software systems. This DoD mandate challenges systems developers to integrate COTS components into systems without compromising the strict reliability and availability requirements of DoD applications. What is more, there are significant strategic and tactical advantages afforded by the ability to adapt quickly to changing situations. These potential advantages challenge developers of DoD systems to find ways to modify and upgrade system components more quickly while reducing the possibility of error.

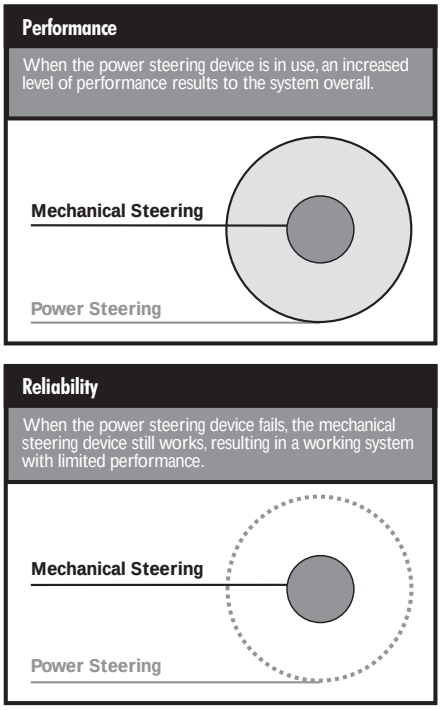
In hardware, problems inherent in the use of COTS components in harsher environments—such as those in which DoD systems operate—can often be solved by packaging. System-level hardware reliability can also be improved by the use of standard fault-tolerance technologies. For example, COTS hardware components can be replicated (replication) and a vote can be taken on their outputs (majority voting). These methods can provide significant protection from hardware faults.

The SEI's work is supported by the Department of Defense. An earlier version of this article appeared in Bridge *(August 1997), a publication of the Software Engineering Institute.*

To ensure the reliability of software is far more difficult. Statistics from the field indicate that software faults cause system failures about 10 times more often than hardware faults [1]. Although a high-assurance software development process can significantly reduce the number of software faults, such processes are typically used only for custom-made software—software designed to one customer's specifications. Most COTS software components, however, are sold as "black boxes" with no warranty and are not typically subject to rigid development, verification, or testing processes. It often is possible to obtain the source code of a COTS software component by paying a large sum of money to the vendor. With the source code, the customer can then subject the COTS components to a high-assurance inspection and testing process and make any modifications that are needed. But once a COTS software component has been modified, it is no longer COTS software, and because the modified COTS software is no longer compatible with the vendor's future releases, most if not all of the benefits of the COTS approach are lost. Therefore, this approach—making proprietary modifications to COTS components—is inconsistent with the original motivation for their use.

Existing architectures cannot tolerate software faults, including faults caused by COTS components or by component changes. This makes the DoD mandate to increase the use of COTS components a challenge to implement. Systems must maintain their existing level of performance even when upgraded components are introduced and do not work under all circumstances. For COTS components to be used safely and effectively, a software fault-tolerant architecture—one that allows developers to modify existing applications and to try out new or upgraded COTS software components easily, affordably, and reliably—is essential.

Figure 1. *Analytically redundant module: a hardware example.*



The Simplex Architecture Solution

Replication and majority voting are effective tools for dealing with random hardware faults. The probability that the majority of replicated hardware components will have the same random fault is extremely small. Unfortunately, replication and majority voting are ineffective against software faults. Given the same inputs, replicated software components will produce the same results, right or wrong.

N-version programming is an approach that is intended to randomize software errors and thus make majority voting work for software faults. In this technique, different programmers build different versions of the same software (or critical parts of a software system) with the idea that different designers and implementers will produce different errors. Therefore, when one system fails under a given set of circumstances, the other probably will not fail. A pragmatic way to use N-version programming is to use different vendors' COTS components with the same interface. For example, in the Boeing 777, three different vendors' Ada run-times and compilers are used [2]. However, because some studies have indicated that some errors will still be shared among the independently developed systems [3], the FAA DO 178B certification process requires that each of the run-times be certified together with the applications. As pointed out by FAA DO 178B, N-version programming may provide some reliability improvement, but the improvement cannot be quantified, and the results cannot be relied on.

Software faults are the result of product complexity that is beyond the developers' capabilities in specification, design, verification, and testing. A well-established engineering approach to guard against the failures of a complex system is to provide a simpler back-up system with assured reliability. For example, the power-assisted steering system in cars is built on and backed up with a simpler mechanical steering system. The two steering systems are not different designs that meet a common specification; the requirements for them are different. One set of requirements emphasizes performance (ease of steering) while the other emphasizes reliability: safe operation even in the presence of engine or hydraulic-system failure. The mechanical steering system is said to be analytically redundant with the power-assisted steering system in the sense that it provides just enough of the power steering system's performance to assure safety (see Figure 1). Power-assisted brakes follow the same principle.

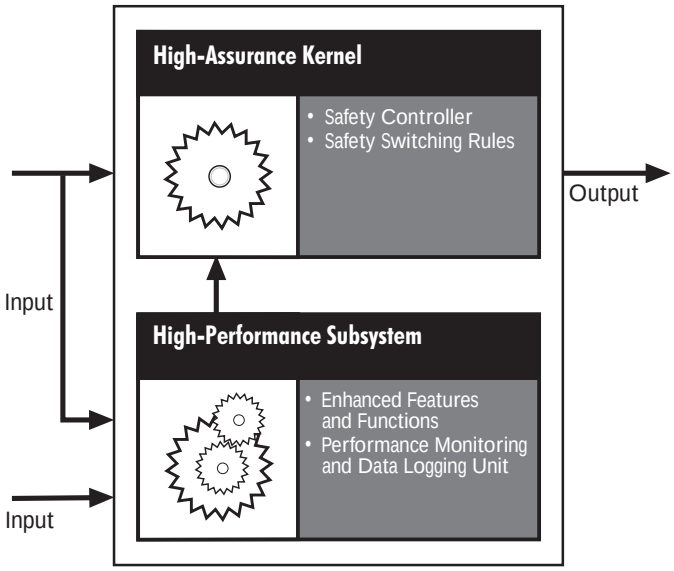
Analytic redundancy can be and has been applied to software systems. Using analytic redundancy, a system is partitioned into a high-assurance portion and a high-performance portion. The high-assurance application kernel is designed to ensure simplicity and reliability. Because of the need to apply costly high-assurance processes to the kernel, the system must be designed such that the rate of changes to the high-assurance kernel is much slower than the rate of changes to the high-performance subsystem. Therefore, COTS components with uncertain reliability are not used in the high-assurance kernel. On the other hand, COTS components *can* be used extensively in the high-performance subsystem. This model is applied in the Boeing 777: a high-assurance backup software controller, known as the secondary digital controller, implements the tried-and-true 747 control laws, whereas a high-performance 777 software controller serves as the normal digital controller [2].

To do the right control job, the high-performance subsystem receives information from a wider variety of sensors compared to the information that the high-assurance kernel receives. The kernel monitors the system state. If the subsystem is driving the system toward a state that the kernel cannot control, the kernel dynamically takes over, meaning that the kernel's outputs are used instead of those of the high-performance subsystem. The kernel can reset and restart the subsystem if and when certain constraints are violated. After the kernel has successfully brought the system back to a new and stable system state, the kernel switches control back to the high-performance subsystem (Figure 2). Since residual software errors are activated only infrequently in certain system states, the subsystem will behave correctly most of the time. This approach works well for systems that have states that can be monitored, such as feedback control and command-and-control applications.

When combined with technologies for real-time computing and component swapping, this approach can also be used to implement upgrades to the high-performance subsystem while the system is on-line. The upgrade need not be perfectly reliable. Failures in upgrades of the high-performance subsystem are no different from the activation of residual errors in the subsystem: The kernel will take over if the new subsystem misbehaves. In addition, the kernel can dynamically return control back to the old version of a component when the upgrade fails, as shown in Figure 3.

Software engineers at the Software Engineering Institute (SEI) have integrated well-established technologies—high-assurance application-kernel technology, address-space protection mechanisms, real-time scheduling algorithms, and

Figure 2. *Run-time replaceable analytically redundant unit.*



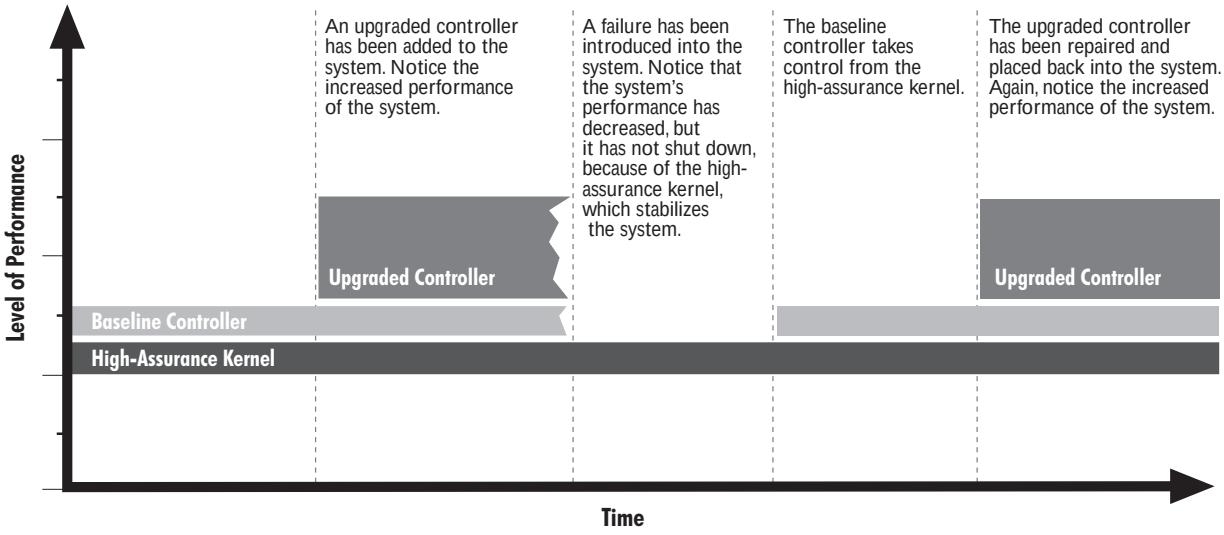


Figure 3. *Analytically redundant module: reliability and performance.*

methods for dynamic communication among modules—to create a framework for the reliable evolution of software systems. This framework is called the Simplex architecture [4]. Although most of the technologies upon which the Simplex architecture is based have existed for some time, the increased adoption of these technologies is making the Simplex architecture increasingly viable.

Under the Simplex architecture, each major system function is implemented as an analytically redundant module consisting of a high-assurance application kernel and a high-performance subsystem, the components of which can be swapped in real time. Like power-assisted steering and power-assisted brakes in a car, analytically redundant software modules can be put together to form an application just as any modules can, except that the components in an analytically redundant module can be replaced easily, reliably, and with no adverse effect on the rest of the system. Should the high-performance portion prove through deployment to be sufficiently reliable, the Simplex architecture also permits users to replace an analytically redundant module with a nonredundant software module consisting only of the high-performance portion. In this way, users can dynamically balance the sometimes conflicting concerns of reliability and efficiency.

In addition to the maintenance of system reliability when COTS software is used, Simplex has proven to be useful for other dependable-system applications. The SEI has participated in several pilot studies that have tested the concepts described in this article in prototypes of real-world applications. These include

- The INSERT (INcremental Software Evolution for Real-Time applications) project: The objective of this project is to generalize and scale up the technologies of the Simplex architecture for dependable evolution of on-board avionics systems. The problem of upgrading mission-control software for the F-16 aircraft is being used as a demonstration vehicle. The sponsors are the Defense Advanced Research Projects Agency Evolutionary Design of Complex Software Programs and the Air Force Research Laboratory (AFRL). The participants are AFRL, Lockheed Martin, Carnegie Mellon University, and the SEI.
- New Attack Submarine Program fault-tolerant submarine control: The objective of this project is to develop, demonstrate, and transition a COTS-based fault-tolerant control system that can be upgraded inexpensively and dependably. The sponsors are the Office of Naval Research (ONR) and Naval Systems Engineering Activity PMS-450. The participants are Naval Surface Warfare Center, Carderock Division, ONR, Electric Boat Corporation, and the SEI.
- CMU's Real-Time Multivariable Control of Plasma-Enhanced Chemical Vapor Deposition project [5]: The objective of this silicon wafer manufacturing project is to demonstrate the use of the Simplex architecture as a basis for the control architecture in manufacturing process-control software. The project is based on a suggestion by engineers from SEMATECH (SEmiconductor MAnufacturing TECHnology). The participants are Carnegie Mellon University and the SEI.

The SEI has also developed demonstration prototypes of the Simplex architecture. The simplest of these demonstration prototypes can be viewed as a QuickTime movie on the SEI Web site (http://www.sei.cmu.edu/technology/simplex/SIMPLEX.MOV). In this demonstration, a feedback-control-loop device controls the positioning of an inverted pendulum. The purpose of the control software is to balance the pendulum in an upright position and keep it as close to the center position as possible. The demonstration shows a safe on-line upgrade from a legacy C program to an Ada 95 program that implements an improved control algorithm. The Ada program visibly im-

proves the control performance. When a bug is introduced into the Ada code and the flawed Ada program is swapped back into the system, the system detects the fault and transfers control back to the C program. The pendulum remains in balance throughout the transfer to the high-performance Ada program, the transfer to the flawed Ada program, and the reversion to the C program. Live interactive demonstrations of more advanced applications such as distributed fault-tolerant controls are available at the SEI for those who wish to pursue this subject further.

For more information about Simplex architecture, visit the SEI Web site at http://www.sei.cmu.edu/technology/dynamic_systems/simplex/introduction/simplex01.shtml.

Summary

It is more important than ever that mission-critical systems be reliable, affordable, and capable of evolving to prevent obsolescence. In this article, we have reviewed a set of existing technologies upon which we can develop an application architecture that is designed to meet these challenges.

The early success of analytically redundant software modules in high-reliability applications provides grounds for optimism that the DoD can achieve the goal of reliable, affordable, evolutionary acquisition of mission-critical systems that exploit the advantages of COTS components. ◆

Acknowledgments

We thank Carol Sledge and John Foreman for their helpful reviews and Mark Paat for creating the graphics for this article.

About the Authors

Lui Sha is a senior member of the technical staff of the SEI, a fellow of the Institute of Electrical and Electronics Engineers (IEEE), an associate editor for the *Real-Time System Journal*, and the chairman-elect of the IEEE Real-Time Systems Technical Committee. He made key contributions to the development and transition of generalized rate monotonic scheduling theory, which has been successfully used in practice and is now supported by the POSIX real-time extension and by Ada 95. He formulated the Simplex architecture and led its development from a concept to the core product of the SEI's Dependable Systems Upgrade Initiative.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
Voice: 412-268-5875
Fax: 412-268-5758
E-mail: lrs@sei.cmu.edu
Internet: http://www.sei.cmu.edu/technology/dynamic_systems/simplex/

John B. Goodenough is the chief technical officer of the SEI and was named a fellow of the Association for Computing Machinery in 1995. He is the former leader of the Rate Monotonic Analysis for Real-Time Systems Project. He was a distinguished reviewer for the Ada 95 language revision effort and has served as head of the U.S. delegation to the International Organization for Standardization working group on Ada. He was the principal author of the document specifying the revision requirements for Ada 95 and has served as chairman of the group responsible for recommending interpretations of the Ada language.

Bill Pollak is a senior writer and editor, member of the technical staff, and team leader of the Technical Communication team at the SEI. He is the editor and co-author of *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems* (Kluwer Academic Publishers, 1993) and has written articles for the *Journal of the Association of Computing Machinery Special Interest Group for Computer Documentation* and *IEEE Computer*.

References

1. Gray, J., "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Transactions on Reliability 39*, Vol. 4, October 1990, pp. 409-418.
2. Yeh, Y.C., "Triple-Triple Redundant 777 Primary Flight Computer," *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Vol. 1, New York, N.Y., Feb. 3-10, 1996, pp. 293-307.
3. Knight, J.C. and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming," *IEEE Transactions on Software Engineering*, SE-12(1):96-109, January 1986.
4. Sha, L., R. Rajkumar, and M. Gagliardi, "Evolving Dependable Real-Time Systems," *Proceedings of the 1996 IEEE Aerospace Applications Conference*, Vol. 1, New York, N.Y., Feb. 3-10, 1996, pp. 335-346.
5. Knight, T.J., D.W. Greve, X. Cheng, and B.H. Krogh, "Real-Time Multivariable Control of PECVD Silicon Nitride Film Properties," *IEEE Transactions on Semiconductor Manufacturing*, Vol. 10, No. 1, February 1997, pp. 137-146.

A Software Development Process for COTS-Based Information System Infrastructure

Part II: Lessons Learned

Greg Fox and Steven Marcom, *TRW*
Karen W. Lantner, *EDS*

Part I of this article (CROSSTALK, March 1998) described the Infrastructure Incremental Development Approach process model. Part II describes a particular application of that model and examines the practical lessons learned and pitfalls encountered.

Modern software developers are guided by a variety of formal and informal processes that help to organize and control development activities across large groups of developers or multiple organizations. These processes supply the discipline and order lacking in many early development efforts. The currently available inventory of documented process methods has a limitation: most assume the system being built will be coded largely from scratch. As a result, the processes do not address many of the challenges associated with building systems that contain large amounts of commercial-off-the-shelf (COTS) software.

The Infrastructure Incremental Development Approach (IIDA) is a combination of the classical development model and the spiral process model to accommodate the needs of COTS-based technical infrastructure development. Each stage of the development cycle is augmented with a series of structured prototypes for COTS product evaluation and integration. This close coupling of prototyping and development stages characterizes the IIDA. The critical success factors for this method are the early establishment of an integrated development environment in which to install the COTS products and early planning for the prototype integration and testing environment, including simulated applications and other test software.

© 1997 IEEE. This material is adapted and reprinted with permission from a paper presented at the IEEE/SEI-sponsored Fifth International Symposium on Assessment of Software Tools and Technologies, Pittsburgh, Pa., June 3-5,1997, pp. 8-10.

Application of IIDA

The following describes experiences using the IIDA method from 1994 to 1997 to develop the initial versions of an infrastructure to support business applications developers for a large, enterprise-wide heterogenous system. The types of COTS products that were integrated included

- Operating systems provided by four different vendors.
- End-user interface COTS software to provide a common graphical user interface.
- Middleware COTS products to provide a uniform transaction processing capability.
- Combinations of COTS software and glue code for specialized services such as security and fail-over recovery.
- Relational database management systems.
- COTS applications for systems management, e.g., software distribution and remote database administration.

Conventional and Unconventional Wisdom

Assumptions at the beginning of the development cycle were that the use of infrastructure COTS products would provide the following benefits.

- Using COTS products would reduce development costs and overall schedule.
- As a corollary, the development cycle would be accelerated.
- Feasibility demonstrations could be put together quickly.
- End-product quality would be higher as measured by a richer feature set and increased system robustness (assuming the selected COTS product is mature) [1].
- COTS vendors would provide maintenance for their COTS products.

The experience of integrating infrastructure COTS products and developed code refocused attention and revealed an additional set of assumptions for future developments.

- Accelerated development catapults you immediately into an integration and test activity.
- Hands-on evaluation requires early simulated applications in an integrated environment; these simulated applications and other test software can represent significant development costs.
- Maintenance on identified problems is provided by the COTS software vendor, but problem investigation and identification by the integrator are the most costly parts of COTS software maintenance.
- Maintenance turnaround time by the vendors can be a significant problem.

Lifecycle Implications

The development method must be specifically tailored to accommodate COTS product integration. This entails a set of assumptions and constraints quite different from custom-built development. Some of the more important of these follow. For a description of the referenced development stages (definition and analysis, functional design, physical design, and construction and test), see Part I of this article.

The front-end processes in the definition and analysis stage must support concurrent requirements and COTS product analysis. The analysis prototype

in the functional design stage must provide for iteration and a flexible linkage between the COTS product evaluations and the feedback loop to requirements analysis.

During the construction stage, the development processes acquire a dual nature when COTS product integration is introduced. One process path is valid for COTS product integration, and another process path is valid for developing the glue code and custom-built components. These two process paths are equivalent but consist of different activities and products. In addition, all COTS products, glue code, and custom-built components must be integrated together to complete development.

During the construction stage, the development of glue code that integrates COTS products and fills in missing functionality is similar to the development of traditional software; the traditional process of coding, unit testing, and integration is applicable.

For COTS products, the construction stage is when COTS products undergo detailed tuning and configuration and when the interfaces and threads between components are exercised in a multi-COTS product environment. COTS product tuning, configuration, and integration have an analog to code and unit-test activities. Unit test with COTS products is "black-box" (vs. "white-box") testing, and the focus is on interfaces and COTS product behavior. For example, unit testing of the transaction processing monitor consisted of exercising all the application programming interface (API) calls supported by the product as configured within the target environment.

Traditional software maintenance activities must be expanded in scope and extended to provide continuing COTS product support. This support starts early in the lifecycle. Application developers must have early deliveries and training for partially completed infrastructure functionality to keep their development lifecycle within reasonable time frames. Developers also require on-site, hands-on direct support from infrastructure developers and integrators to ensure acceptance and proper use of the infrastructure products.

Configuration control must be organized and in place early to accommodate multiple versions of the COTS products and configuration files. Separate environments for development and integration must be well-defined and structured to accept the delivered COTS products. Early support for multiple baselines must be in place as the combinations of COTS products become complex.

Throughout the lifecycle, feedback loops allow ongoing re-evaluation of the COTS products. Analysis prototypes (functional design stage) determine feasibility of a COTS-based solution and provide feedback to the requirements definition (definition and analysis stage). Design prototypes (physical design stage) provide hands-on experience with potential COTS products and feedback to the COTS product selection process (functional design stage). Detailed design prototypes (physical design stage) exercise functionality of selected COTS products, verify adherence and consistency with design expectations, reveal detailed behavior and performance characteristics, and give insight into the invocation parameters. The demonstration prototype (construction and test stages) is used to unit test the COTS products using black-box testing to simulate application behavior or environment. Each stage is a potential source of feedback to previous stages.

Practical Considerations

The following practical considerations were encountered during two years of experience using the IIDA.

- The COTS product integrator does not develop the COTS product but still must internally know it. The integrator must understand the complete set of capabilities provided by the COTS product to select the appropriate subset of capabilities based on application developer needs for a given release of infrastructure. The integrator must understand the limitations and nuances of the COTS product to exercise it. For example, does it run on all of the required platforms? Does it operate the way it is intended? Does it have a heritage from a different paradigm (PC vs. UNIX workstation)?
- The system administrators and configuration management staff need to know how to configure the COTS products. Few complex COTS products work straight out of the box. To support early prototypes and evaluations, not only do the designers and developers need to understand the products, the development system administrators need to understand how to install and manage the product configuration. In addition, configuration management needs to understand how to configure the product versions.
- "COTS castles are often built on the sand of configuration files." Configuration files and data can be as complex as code. They must be understood. For example, a transaction processing monitor configuration file is inherently complex; training is required to know how to use it. Configuration files can be site-specific and require a strategy to manage files for different sites including site-specific parameters, implementation requests, and file distribution.
- When installing infrastructure components in new sites, the following documents that are not part of normal lifecycles are critical for the configuration of COTS products.
  - Release notes (installation guidelines, operational parameters, tuning guidelines, etc.)
  - Site configuration guidelines (guidelines to help site designers choose appropriate hardware and software suites and rules for scaling and resource allocation).
- Version compatibility between COTS products, the operating system, and glue code is critical. This also applies to different sites including the external integration and test function. Software problems and nuances of use discovered during integration are not necessarily embedded in selected COTS products

but often derive from specific characteristics of operating system versions or communications protocols. If application developers, infrastructure developers, and test sites are allowed to independently manage their computing platform configurations (including operating system and database management system), trouble-shooting infrastructure anomalies is extremely difficult.

- Licensing adds a dimension of complexity and needs to be worked with early. Issues include the number and types of licenses required for the environment. Short-term COTS evaluation licenses need to be managed, and transition needs to be planned from evaluation to product license. Procurement of production licenses within government agencies can require a long lead time and needs to start early with the Bill of Materials (BOM).

Technical Management Considerations

The following considerations can be easily overlooked during the planning cycle.

- The development facility including hardware, development tools, and configuration management must be ready to go before the first COTS product arrives for prototyping. Facility readiness fuels the accelerated development that using COTS products can provide but moves the requirement for a fully implemented development facility to early in the effort. Determining COTS suitability requires a realistic target configuration with a strong system administration team in place from the start.
- The BOM represents the contract for COTS products and versions. It is required early for field development sites and is essential for successful deployment.
- Technology infusion occurs by virtue of COTS product upgrades whether it is planned or not. Product upgrades can occur during any phase of the lifecycle. Allowing for technology infusion can exploit new potential products on the market.
- The investment in training is a significant but often overlooked cost of using COTS products. Management needs to plan for the expertise of individuals to be shared across organizations. In particular, field sites need training, especially in system administration.

Conclusion

Integration with COTS software products requires adjustment and accommodations to the development approach vs. traditional software development. Preparations must be made to start prototyping activities and integration activities immediately to exploit COTS product advantages and accelerate development. Additional resources must be allocated for late in the development cycle to provide maintenance and support to the user community, i.e., the application developers. ◆

Acknowledgment

We thank David P. Maloney, a software development manager at TRW, for his contributions to this article. Many of the insights in the application of IIDA resulted from his work.

About the Authors

Greg Fox is a TRW Systems Integration Group technical fellow and the director of technology for the Information Services Division. He has 28 years experience in mostly large or complex information systems. He has led the architecture development and system integration for several large COTS-based systems and has been TRW's information systems infrastructure project manager and chief architect for the Integration Support Contract for Internal Revenue Service (IRS) modernization. He has engineering degrees from Massachusetts Institute of Technology and University of Southern California and has published over a dozen papers.

TRW, Inc.
MVA1/4943
12900 Federal Systems Park Drive
Fairfax, VA 22033
Voice: 703-876-4396
E-mail: greg.fox@trw.com

Steven Marcom is a senior systems analyst with the TRW Information Services Division. He has 30 years managerial and technical experience developing computer systems for civil government, defense, and commercial customers. He was TRW's systems lifecycle deputy manager and information systems infrastructure process engineer for the Integration Support Contract for IRS modernization. He has been active in Rapid Application Development, COTS integration, and prototyping activities. He has a bachelor's degree from Pomona College and a master's degree from the American University of Beirut, both in mathematics. He teaches software development and integration at TRW.

TRW, Inc.
FP1
12900 Federal Systems Park Drive
Fairfax, VA 22033
Voice: 703-803-4814
E-mail: marcoms@gisdbbs.gisd.trw.com

Karen W. Lantner is a program/project manager for EDS in New York City. She has 24 years management and technical experience, during which she has managed and consulted on large federal software development and COTS integration projects. A member of the team that developed the EDS Systems Life Cycle Methodology, she continues to have a special interest in software development methods. She has a bachelor's degree and a master's degree from Brown University.

EDS
A5N-B50
13600 EDS Drive
Herndon, VA 22071
Voice: 800-336-4498, box no. 52032
E-mail: karen.w.lantner@aexp.com

Reference

1. Langley, R.J., "COTS Integration Issues, Risks, and Approaches," *SIG Technology Review*, TRW Systems and Integration Group, Vol. 2, No. 2, Winter 1994, pp. 4-14.

Three Dimensions of Process Improvement
Part III: The Team Process

Watts S. Humphrey
Software Engineering Institute

Part I of this article (CROSSTALK, February 1998) described the Capability Maturity Model (CMM)®, and Part II (CROSSTALK, March 1998) addressed the Personal Software Process (PSP)SM. The CMM provides an overall framework that has helped many organizations improve their performance and the PSP shows engineers how to use process principles in doing their personal work. Part III describes the Team Software Process, which shows integrated product teams how to use these processes to consistently produce quality products on aggressive schedules and for their planned costs.

Building a Supportive Team Environment: The Team Software Process

The Team Software Process (TSP) extends and refines the CMM and PSP methods to guide engineers in their work on development and maintenance teams. It shows them how to build a self-directed team and how to perform as an effective team member. It also shows management how to guide and support these teams and how to maintain an environment that fosters high team performance. The TSP has five objectives:

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of three to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

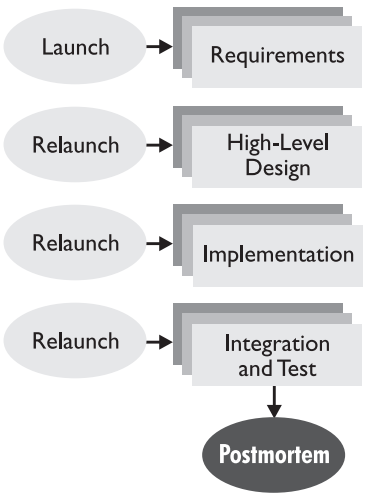
The SEI's work is supported by the Department of Defense. Capability Maturity Model and CMM are registered with the U.S. Patent and Trademark Office. Personal Software Process, PSP, Team Software Process, and TSP are service marks of Carnegie Mellon University.

The principal benefit of the TSP is that it shows engineers how to produce quality products for planned costs and on aggressive schedules. It does this by showing engineers how to manage their work and by making them owners of their plans and processes.

Team-Building Strategies Are Not Obvious

Generally, when a group of engineers starts a project, they get little or no guidance on how to proceed. If they are lucky, their manager or one or two of the experienced engineers will have worked on well-run teams and have some ideas on how to proceed. In most cases, however, the teams have to muddle through a host of issues on their own. Following are some of the questions every software team must address.

Figure 1. *TSP structure.*



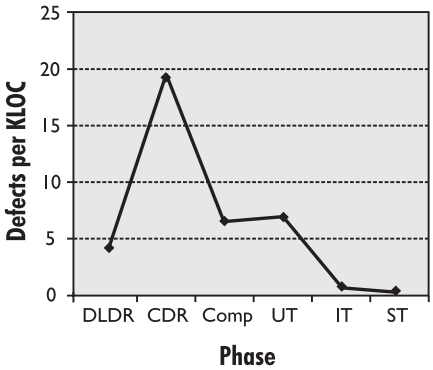


Figure 2. *Defects per thousand lines of code (KLOC) removed by phase.*

- What are our goals?
- What are the team roles and who will fill them?
- What are the responsibilities of these roles?
- How will the team make decisions and settle issues?
- What standards and procedures does the team need and how do we establish them?
- What are our quality objectives?
- How will we track quality performance, and what should we do if it falls short?
- What processes should we use to develop the product?
- What should be our development strategy?
- How should we produce the design?
- How should we integrate and test the product?
- How do we produce our development plan?

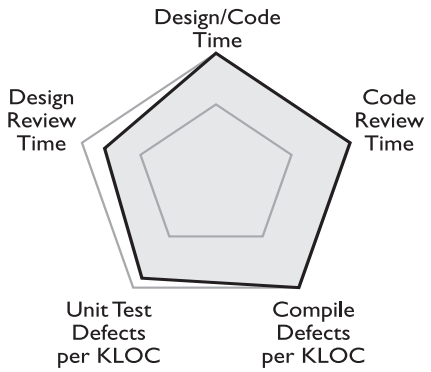


Figure 3. *Component 7 quality profile.*

- How can we minimize the development schedule?
- What do we do if our plan does not meet management's objectives?
- How do we assess, track, and manage project risks?
- How can we determine project status?
- How do we report status to management and the customer?

Most teams waste a great deal of time and creative energy struggling with these questions. This is unfortunate, since none of these questions is new and there are known and proven answers for every one.

The TSP Process

The TSP provides team projects with explicit guidance on how to accomplish their objectives. As shown in Figure 1, the TSP guides teams through the four typical phases of a project. These projects may start or end on any phase, or they can run from beginning to end. Before each phase, the team goes through a complete launch or relaunch, where they plan and organize their work. Generally, once team members are PSP trained, a three-day launch workshop provides enough guidance for the team to complete a full project phase. Teams then need a two-day relaunch workshop to kick off each of the second and each of the subsequent phases. These launches are not training; they are part of the project.

The current TSP version uses 23 scripts, 14 forms, and three standards. The TSP scripts define 173 launch and development steps. None of the steps is complex, but each is defined in enough detail so the engineers can see how to do what they have to do. These scripts guide the teams through the steps of launching and running their projects.

The TSP Launch Process

To start a TSP project, the launch process script leads teams through the following steps.

- Review project objectives with management and agree on and document team goals.
- Establish team roles.
- Define the team's development process.
- Make a quality plan and set quality targets.
- Plan for the needed support facilities.
- Produce an overall development strategy.
- Make a development plan for the entire project.
- Make detailed plans for each engineer for the next phase.
- Merge the individual plans into a team plan.
- Rebalance team workload to achieve a minimum overall schedule.
- Assess project risks and assign tracking responsibility for each key risk.

In the final launch step, the team reviews their plans and the project's key risks with management. Once the project starts, the team conducts weekly team meetings and periodically reports their status to management and to the customer.

In the three-day launch workshop, TSP teams produce

- Written team goals.
- Defined team roles.
- A process development plan.
- The team quality plan.
- The project's support plan.
- An overall development plan and schedule.
- Detailed next-phase plans for each engineer.
- A project risk assessment.
- A project status report.

Early TSP Results

While the TSP is still in development and has only been used with 10 industrial and three student teams, the early results are encouraging. One team at Embry Riddle Aeronautical University (ERAU) removed over 99 percent of development defects before system test entry. Their defect-removal profile is shown in Figure 2.

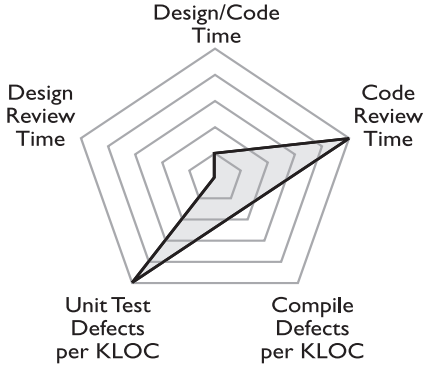
TSP teams also gather the data they need to analyze component quality before integration and system testing. This is done with the aid of the component quality profile, which shows five quality parameters in a bullseye format. With a profile that nearly fills the entire bullseye, as in Figure 3, quality is judged to be good. A profile like that in Figure 4, however, indicates likely problems. The five profile dimensions are shown in Table 1 and explained below.

The data for Figures 3 and 4 came from two of the ERAU team's components. Component 7 had no integration or system test defects, and Component 9 had one integration defect. As can be seen from Figure 4, the development work for Component 9 had inadequate design time, no design review time, and high compile defects. The only surprise is that this component had only one defect in integration test and none in system test.

The component quality profile is based on the following criteria.

- Design time is greater than 50 percent of coding time.
- Design review time is greater than 50 percent of design time.
- Code review time is greater than 50 percent of coding time.
- Compile defects are less than 10 per KLOC.
- Unit test defects are under five per KLOC.

Figure 4. *Component 9 quality profile.*



Dimension	Meaning
Design/Code Time	The ratio of detailed design time to coding time.
Code Review Time	The ratio of code review time to coding time.
Compile Defects/KLOC	The defects per KLOC found in compile.
Unit Test Defects/KLOC	The defects per KLOC found in unit test.
Design Review Time	The ratio of detailed design review time to detailed design time.

Table 1. *Component quality profile dimensions.*

When a factor meets or exceeds these criteria, that profile dimension is at one or on the edge of the bullseye. When the criteria are not met, say 25 percent design review time instead of 50 percent, that dimension value would be one-half or halfway to the center of the bullseye. Once teams gather enough of their own data, they should establish the profile criteria that work best for them.

With TSP data, engineers can determine which components are most likely to have defects before they start integration and system testing. By reworking these defect-prone components before test entry, they can save a substantial amount of test time and produce higher-quality products.

How the TSP Helps Teams Behave Professionally

Perhaps the most powerful consequence of the TSP is the way it helps teams manage their working environment. The most common problem product teams face is unreasonable schedule pressure. Although this is normal, it can also be destructive. When teams are forced to work to unreasonable schedules, they are unable to make useful plans. Every plan they produce misses the edicted schedule and is therefore unacceptable. As a result, they must work without the guidance of an orderly plan and will generally take much longer to complete the project than they otherwise would.

The TSP team's responsibility is to plan and produce a quality product as rapidly and effectively as they can. Conversely, it is management's responsibility to start projects in time to finish when needed. When similar projects have taken 18 months and management demands a nine-month schedule, this is clearly unrealistic. Where was management nine months ago when the project should have started? Although the business need may be real, the team's schedule is only part of the problem. Under these conditions, it is essential that management and the team work together to rationally determine the most effective course of action. This will often involve added resources, periodic replanning, or early attention to high-risk components.

While TSP teams must consider every rational means for accelerating their work, in the last analysis, they must defend their plan and resist edicts that they cannot devise a plan to meet. If management wants to change job scope, add resources, or suggest alternate approaches, the team will gladly develop a new plan. In the end, however, if the team cannot produce a plan to meet the desired schedule, they must not agree to the date. So far, most TSP teams have been able to do this. Teams have found that the TSP provides them convincing data to demonstrate that their plans are aggressive but achievable.

The TSP Manager-Coach

Perhaps the most serious problem with complex and challenging work is maintaining the discipline to consistently perform at your best. In sports and the performing arts, for example, we have long recognized the need for skilled trainers, conductors, and directors. Their job is to motivate and guide the performers and also to insist that everyone meet high personal standards. Although skilled players are essential, it is the coaches who consistently produce winning teams. There are many differences between software and athletic or artistic groups, but they all share a common need for sustained high performance. This requires coaching and support.

Software managers have not traditionally acted as coaches, but this is their role in the TSP. The manager's job is to provide the resources, interface to higher management, and resolve issues. But most important, the manager must motivate the team and maintain a relentless focus on quality and excellence.

Table 2. *PSP and TSP coverage of CMM key process areas.*

Level	Focus	Key Process Area	PSP	TSP
5 Optimizing	Continuous Process Improvement	Defect Prevention	X	X
		Technology Change Management	X	X
		Process Change Management	X	X
4 Managed	Product and Process Quality	Quantitative Process Management	X	X
		Software Quality Management	X	X
3 Defined	Engineering Process	Organization Process Focus	X	X
		Organization Process Definintion	X	X
		Training Program		
		Integrated Software Management	X	X
		Software Product Engineering	X	X
		Intergroup Coordination		X
		Peer Reviews	X	X
2 Repeatable	Project Management	Requirements Management		X
		Software Project Planning	X	X
		Software Project Tracking	X	X
		Software Quality Assurance		X
		Software Configuration Management		X
		Software Subcontract Management		

This requires daily interaction with the team and an absolute requirement that the process be followed, the data gathered, and the results analyzed. With these data, the manager and the team meet regularly to review their performance and to ensure their work meets their standards of excellence.

Conclusion

The CMM, PSP, and TSP provide an integrated three-dimensional framework for process improvement. As shown in Table 2, the CMM has 18 key process areas, and the PSP and TSP guide engineers in addressing almost all of them. These methods not only help engineers be more effective but also provide the in-depth understanding needed to accelerate organizational process improvement.

The CMM was originally developed to help the Department of Defense (DoD) identify competent software contractors. It has provided a useful framework for organizational assessment and a powerful stimulus for process improvement even beyond the DoD. The experiences of many organizations show that the CMM is effective in helping software organizations improve their performance.

Once groups have started process improvement and are on their way toward CMM Level 2, the PSP shows engineers how to address their tasks in a professional way. Although relatively new, the PSP has already shown its potential to improve engineers' ability to plan and track their work and to produce quality products.

Once engineering teams are PSP trained, they generally need help in applying advanced process methods to their projects. The TSP guides these teams in launching their projects and in planning and managing their work. Perhaps most important, the TSP shows managers how to guide and coach their software teams to consistently perform at their best. ◆

Acknowledgments

I thank all the many people who have participated in this work. I especially thank Jim Over for his invaluable support and assistance with both the PSP and TSP work and this series of articles. I also thank Linda Parker Gates, Tom Hilburn, Alan Koch, Mike Konrad, Mark Paulk, Bill Peterson, and Dave Zubrow for their many helpful comments and suggestions.

About the Author

Watts S. Humphrey is a fellow at the Software Engineering Institute (SEI) of Carnegie Mellon University, which he joined in 1986. At the SEI, he established the Process Program, led initial development of the CMM, introduced the concepts of Software Process Assessment and Software Capability Evaluation, and most recently, the PSP and TSP. Prior to joining the SEI, he spent 27 years with IBM in various technical executive positions, including management of all IBM commercial software development and director of programming quality and process.

He has master's degrees in physics from the Illinois Institute of Technology and in business administration from the University of Chicago. He is the 1993 recipient of the American Institute of Aeronautics and Astronautics Software Engineering Award. His most recent books include *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software Process* (1997).

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 412-268-6379
E-mail: watts@sei.cmu.edu.

Telos® Corporation Achieves CMM Level 4

Telos Corporation recently achieved Software Engineering Institute (SEI) Capability Maturity Model (CMM) Level 4 rating for software engineering process maturity in support of the U.S. Army Software Engineering Center (SEC) in Lawton, Okla. Their Level 4 rating places Telos in the top 1.5 percent of the more than 600 CMM-appraised organizations in the software industry.

This achievement is the culmination of a six-year effort on the part of Telos and the Communications Electronics Command Software Engineering Center. The appraisal was conducted by a team consisting of personnel from Telos, SEC Fire Support Software Engineering, the SEI, and Lockheed Martin.

Telos has actively followed SEI guidelines since 1990 because they contribute to increased product quality, improved software performance, more predictable development schedules, and reduced system lifecycle costs. As part of its software process improvement, Telos implemented a comprehensive development environment with a standardized software design method and formed integrated project teams comprised of software engineers, system analysts, programmers, and test and training personnel.

Telos Corporation specializes in network-based solutions for governments and industry worldwide. The company's Fort Sill, Okla. office has fielded nearly 100 major fire support system software versions and now maintains nearly 9 million lines of tactical and support systems code and thousands of pages of documentation. Telos also provides data integration, network security, network and systems integration, and unique products including wireless networks, training and simulation systems, and message handling systems. Telos headquarters are in northern Virginia's Netplex area.

Internet: http://www.telos.com

How Scalable Are CMM Key Practices?

Rita Hadden
Project Performance Corporation

Many software practitioners are convinced that cost outweighs benefit when the Software Engineering Institute Capability Maturity Model's (CMM) key practices are applied to small projects. Practitioners believe that only the complexity that typically comes with project size justifies the investment. This article is based on observations and experience with over 50 small projects. It describes using a repeatable and disciplined approach for system development efforts in spite of short durations. It illustrates the use of professional judgment to appropriately scale down and apply CMM key practices to make a difference in the outcome of small software projects.

Many software professionals believe that the key practices of the Software Engineering Institute (SEI) CMM are inappropriate and not scalable for small software projects (three to four months with five or fewer staff members). These practitioners are convinced that the costs outweigh the benefits unless the complexity that typically comes with project size justifies the investment.

Some professionals also believe that customer satisfaction is their most important measure of performance, meaning that to accept one more requirement is almost always "the right thing to do," no matter how close the production or release date might be. They are convinced that in the "real world," requirements cannot be frozen or "baselined" because there are too many legitimate reasons for requirements to change, and new ones must be added to the current version of the software.

Many developers frequently do not see the benefits of managing requirements, nor do they feel the need to take time for project planning, tracking, and oversight, as advocated by the CMM. They do not understand the payback from a software quality assurance function or configuration management. I recently worked with several managers and practitioners who held these views.

No Defined Requirements But a Set End Date — An Example

The project was to develop a software system that provides a consistent estimation approach and addresses software size, effort, and assumptions for planning. The system's goal was to improve the client organization's software estimation process. The organization's overall strategy was to use the SEI CMM Repeatable level (Level 2) key practices to move toward a more disciplined software development environment. When I started, no requirements had been defined for this estimation system but an end date had been set—the system had to be operational in two months.

The client was an information systems business unit in a large company. A formal CMM-based software process assessment conducted a month prior to my arrival found that this organization's estimation process for software change requests was ad hoc and inconsistent. This estimation process frequently produced understated effort estimates, perhaps due to institutional pressure to get things done faster or a desire to minimize what is involved. In turn, these understated estimates often led to budget overruns, crisis management, eleventh-hour firefighting, low employee morale, and high rates of post-release defects.

My immediate clients, all information systems professionals, had been working on improving their software management practices for over 18 months. Their corporate direction was to use the CMM to guide their process improvement. Some of them had been promoting the CMM for close to two years. The sponsor of our software project was the head of the Software Engineering Process Group. The project team consisted mostly of client personnel to the project part time, and three full-time outside consultants: a junior analyst, a Visual Basic programmer, and me.

Getting Buy-In

A fellow team member and I began the project by meeting with our sponsor and key stakeholders to define the business objectives, scope, and constraints of the estimation system. After reviewing existing relevant documentation, we developed a high-level project plan. It was an ideal opportunity to apply the CMM key practices to our day-to-day project work. In particular, I believed the management and technical practices from Levels 2 and 3 of the CMM were appropriate. These included requirements management; project planning, tracking, and oversight; quality assurance (QA); configuration management (CM); software product engineering; and peer reviews. I recommended to our clients that we apply scaled-down CMM key practices to meet the needs of our estimation system project.

"The system must be finished in two months. We need to start coding right away!" was my sponsor and other team members' reply.

I pointed out that automating a more disciplined software estimation process with a structured and repeatable approach would not only assist their

overall process improvement effort but also would help the current project meet its pressing deadline as well.

As far as I could tell, I had been brought into this project for three reasons. First, I had earned the trust and respect of some of the key players in this organization by contributing to a previous process improvement project. Second, I was a seasoned software engineer and manager with over 70 software projects from which to draw. Finally, I was certified by the SEI as a lead software process assessor.

My sponsor agreed to let me apply some of the CMM key practices on the grounds that they would legitimize the product we were developing. I knew I was going to have to make believers out of these skeptics.

Turning Expectations into Requirements

Our stakeholders were software project managers, team leaders, functional subject matter experts called "evaluators," software practitioners who performed estimation called "estimators," and coordinators of the many software change requests. These stakeholders were accountable for the effectiveness and efficiency of this organization's estimating process.

My team of one analyst, one developer, and two part-time client functional experts had to first come to consensus on "what is a requirement?" We agreed that a requirement is a functional or technical capability needed to solve a problem or achieve an objective. A good requirement is traceable to business objectives and related system lifecycle components. It is consistent with the scope and constraints of the product, incorporates stakeholder expectations, is measurable against acceptance criteria, and is maintainable over the product's lifecycle.

To expedite the requirements-gathering phase, my team developed a set of straw man functional expectations. These expectations were based on our documentation review, our understanding of the organization's estimation process and user desires, our knowledge of industry best practices, and the objectives, scope, constraints, and assumptions of the estimation system.

Next, we used the straw man set of expectations and Joint Application Design techniques to elicit additional input from our stakeholders. We held focus groups and one-on-one interviews. We then analyzed all gathered expectations and developed a manageable set of functional requirements. Where necessary, we consolidated similar expectations and identified and resolved conflicts between expectations.

Explicit vs. Implicit Requirements

My team then supplemented these "explicit" requirements with "implicit" technical requirements. Implicit requirements include stakeholder expectations that were not articulated but are essential to develop a product that achieves exceptional user satisfaction. For example, we ensured that the system's response time and maximum number of concurrent users were stated as measurable requirements. We also addressed usability, availability, security, maintainability, and portability of the estimation system.

We documented all requirements and validated them with all stakeholders. We requested that our key stakeholders review and sign off on the Statement of Requirements for the estimation system, then baselined these requirements. We obtained the sign-off three weeks after the project start.

Building In Quality

Based on the system requirements, we next developed a conceptual design that we revalidated with our stakeholders in small walk-through sessions where feedback and ideas were solicited. This conceptual design was informally documented in two days. Three concurrent efforts followed: one to create a detailed software development plan (SDP), a second to design and prototype user interface data entry screens and system navigation using a Rapid Application Development, and a third to design database structures and define data validation criteria. We involved our stakeholders throughout the process.

Using Requirements to Drive the Software Size Estimate

We now had what we needed to get a relatively reliable estimate of the size of the estimation system. We used function points and work breakdown structures to estimate size. The results from these methods took less than two days to produce and were remarkably close (4.3 staff months for the construction phase using work breakdown structures vs. 4.7 using function points). We met with senior management and presented our estimates of size, effort, cost, schedule, critical computer resources, and what it would take to do the project "right the first time." Faced with the detailed planning data, senior management had to choose among cutting scope, changing the schedule, or adding more people to the project. They agreed to allocate more resources and time: one programmer for four weeks and one additional elapsed month.

The SDP also contained a detailed task plan, test plan, QA plan, CM plan, measurement plan, and risk management plan. Because this was a small project (2,275 hours, [3.5 months]), we scaled our SDP accordingly to 28 pages. To save time, we rewrote an existing QA plan to meet our needs.

Involving Stakeholders and Controlling Requirement Changes

Five weeks into the project, our designs and prototype were ready for a walk-through with our stakeholders. Many issues and concerns surfaced at this session that we had to address. In addition, many requirements changes were proposed following the walk-through. We listened to each rationale for change and recorded the request. Where a change improved usability and the effort was negligible, we included it in our revised design. Dealing with these challenges upfront minimized rework for us later.

We resisted the temptation to add a new feature for every stakeholder concern. We presented the total requested change in scope and its associated impact on schedule, resource, and cost to the client senior management and Software Configuration Control Board. It

would have taken 31 additional staff days to incorporate all the requested changes.

The stakeholders decided to revisit the change requests in a subsequent release of the system. Our requirement control approach helped us gain buy-in and credibility from all project participants.

Paving the Way for Culture Change

Because senior management recognized that this project would require many people to fundamentally change the way they perform software estimation, we wanted to pave the way for this culture change. To meet this objective, we deployed a set of spreadsheets that emulated the new estimation process six weeks prior to piloting the new estimation system. With these spreadsheets, we trained future users of the estimation process in the "whys" and "hows" of the new procedures.

We left the detailed design of the reporting requirements until last. Key stakeholders were invited to sit by our side as we designed the reports and queries that contributed to their job effectiveness. We also requested stakeholder participation and feedback in our system testing to fine-tune our system. Meanwhile, we developed training materials and user documentation that emphasize audience participation and usability. Later, we provided multiple training sessions that gave participants hands-on system experience.

Tracing Test Cases to Requirements

As soon as we received approval for our design, we began to define test cases and expected results for each of the system functions. We had limited time for testing, so we wanted to ensure each test case was traceable back to one or more requirements and that our product met the stated requirements.

We then created a test database and seeded it based on our test cases. As each software module was completed, we performed functional testing using the test cases and seeded database. We compared actual results against our documented expected results for each test case. At the end of each testing session, we reviewed our test results with the responsible software developer. Hundreds of defects were uncovered early enough to correct with minimum effort. Some modules were particularly problematic and had to undergo several iterations of functional testing.

Keeping Our Project on Track

Our team met weekly to discuss status, issues, and required actions. I worked continuously to mitigate the risks to project success. These risks included a language barrier between the two programmers, since one spoke little English; insufficient face-to-face communication with users of the system, which could result in misunderstanding of the new estimation process; performance and usability issues that could cause stakeholders to not fully accept the product; delays in functional testing due to defects, which could jeopardize the start of system test and overall project schedule.

We also met periodically with the client organization's QA, CM, and senior management. QA's role was to review and monitor our design and development activities to verify compliance with the organization's standards and procedures. CM's function was to maintain the integrity of our product through configuration identification, change control, status accounting, and audit. CM also ensured that all requested changes to the functionality of the estimation system were reviewed by senior management and the Software Configuration Control Board.

All this oversight and coordination set the stage for an extremely smooth pilot. This pilot lasted a week and involved a dozen users working hard to exercise all aspects of the estimation system. These users entered "real-world" data from recent estimates they had prepared. They also tested the system's user's guide. Based on their feedback, we enhanced the user's guide to include exception handling. No rework was required on the system software.

The estimation system was accepted with enthusiasm and went into full production without delay. No significant defects had been identified to date six months following the rollout.

CMM Key Practices Are Scalable!

We held a feedback session to develop lessons learned with members of our team when the project was over. To our surprise, we heard that this was the *first* time that a PC-based system developed by this group had been delivered on time, within budget, and with satisfied stakeholders. Even the Visual Basic programmers admitted that a managed set of requirements, a documented design, early and frequent user involvement, and disciplined approach to testing contributed to the success of the project.

These practitioners' beliefs that design was "paperwork," that there was no time for walk-throughs, and that ad hoc testing was sufficient were beginning to change. Their conviction that requirements could not be baselined was also becoming less absolute. We had given them an alternate way to approach software development—we had made the CMM come alive for them!

Looking back, I feel tremendous satisfaction for sticking to a repeatable and disciplined approach for our system development effort in spite of its short duration. I can imagine the outcome of this project had we given in to pressure to start coding at the beginning of the project.

Using professional judgment to appropriately scale down and apply CMM key practices instead of using "code and load" did make a difference. Managing requirements helped us control "scope creep" and keep our stakeholders aware of the trade-offs, allowing them to make informed decisions. Using a disciplined process helped us discover defects prior to testing, minimize rework, and reduce post-release defects. We delivered a better product on

see CMM, page 23

Software Certification Laboratories
To Be or Not to Be Liable?

Jeffrey Voas
Reliable Software Technologies

Software certification laboratories (SCLs) will potentially change the manner in which software is graded and sold. However, a main issue is who is liable when, during operation, certified software acts in a manner that the SCL certified was not possible. Given software's inherently unpredictable behaviors, can SCLs provide precise-enough predictions about software quality to reduce their liability from misclassification to a reasonable level? SCLs' survival and effectiveness largely depend on solving the issues of certification liability.

If you visit a doctor's office, you will often hear terms such as "independent laboratory," "second opinion," "additional tests," or "colleague consultation." What these amount to is a doctor getting another party or process involved in a diagnosis or treatment decision. Doctors use outside authorities, in part, to reduce the risk of malpractice. The more consensus built with respect to a particular course of action, the more due diligence has been shown. And more parties are culpable if something goes wrong. For instance, if a medical laboratory returns a false diagnosis that a tissue sample is cancerous and the doctor begins treatments that were not necessary, the doctor can ascribe some or all of the liability for this mistake to the laboratory. The added costs from spreading liability around in this manner are one reason for the cost increases in health care. Each extra opinion and extra test increase patient costs, because each care provider is a malpractice target.

A Demand for Independent Certification

In the software world, a similar phenomenon is being observed. Software producers and consumers are more frequently demanding that independent agencies certify that programs meet certain criteria. Vendors prefer to not be responsible for guaranteeing their software, and software consumers want unbiased assessments that are not based on a sales pitch. Incredible as it may seem, vendors, who typically "cut all corners" in costs, are willing to pay the costs associated with placing this responsibility on someone else.

Because of the demands for SCL services, business opportunities exist for organizations that wish to act in this capacity. By paying SCLs to grant software certificates, independent software vendors (ISVs) partially shift responsibility onto the SCL for whether the software is "good." The question is whether this method of liability transfer will be as successful in software as it has been in health care. As we will discuss, if SCLs set themselves up right, they can build more protection around themselves than you might think, leaving the ISV holding a "hot potato."

There are several relatively obscure SCLs in existence today, e.g., KeyLabs, which handles applications for 100 percent pure Java. Other than these small, specialized laboratories, the next closest match to an SCL (conceptually speaking) is Underwriter's Laboratory (UL). UL certifies electrical product designs to ensure that safety concerns are mitigated. Rumors are that UL is interested in performing SCL services, but to my knowledge, UL has not yet become an SCL.

Commercial software vendors are not the only organizations that see the benefit of SCLs. NASA felt the need for standardized, independent software certification for the software they write and purchase. NASA now has an SCL—the Independent Verification and Validation Facility in Fairmont, W. Va. Intermetrics is the prime contractor at the facility, and their job is to oversee the certification process and provide the necessary independence. This SCL provides NASA with a common software assessment process over all software projects (as opposed to each NASA center performing assessments in different ways). The NASA facility certifies software developed both by NASA employees and NASA's contractors.

The beauty of having SCLs is that they provide a quasi-fair "playing field" for all software vendors—each product is supposed to be given equal treatment. The issue is that when software fails in the field, and an independent party provided an assessment that suggested that the software was good, does the independent party bear any responsibility for the failure?

Who Is Liable When Software Fails?

Who is liable when certified software fails—the ISV, the SCL, both, or neither? More specifically, how is liability divided between these groups? First, the question of how much liability, if any, can be placed onto the SCL will be addressed. By figuring out the liability incurred by an SCL for its professional opinions, we can determine how much liability is offloaded from the ISV.

Limiting Liability

SCLs stand as experts, rendering unbiased professional opinions. This exposes SCLs to possible malpractice suits. Schemes to reduce an SCL's liability include insurance, disclaimers on validity of the test results, and SCLs employing accurate certification technologies based on objective criteria. Of these,

the best approach is to only certify objective criteria, and avoid trying to certify subjective criteria.

Subjective vs. Objective Criteria

Different software criteria can be tested by SCLs, spanning the spectrum from guaranteeing correctness to counting lines of code. Subjective criteria are imprecise and prone to error. Objective criteria are precise and less prone to error. For example, deciding whether software is correct is subjective because of the dependence on the precise definition of "correctness." SCLs should avoid rendering professional opinions for criteria that are as contentious as this.

Instead, SCLs should assess objective software characteristics such as exception handling calls and lines of code. Testing for objective criteria is not rocket science. Troubles will begin, however, when an SCL tries to get into the tricky business of estimating a subjective criterion such as software reliability. By only certifying objective criteria, the chances of inadvertent favoritism for one product over another is reduced.

The ICSA Certification Approach

The International Security Computer Association (ICSA) is a for-profit SCL that has taken an interesting approach to the liability issue. They use industry consensus building. ICSA only certifies that specific known problems are not present in an applicant's system. This is an objective criterion. Their firewall certification program is based on the opinions of industry representatives who periodically decide for which known potential problems the software should be checked. Over time, additional criteria are introduced into the certification process. This adaptive certification process serves two purposes: it adds rigor to the firewall certification process and produces a steady stream of business for the ICSA. To further reduce liability, ICSA does not claim that their firewall certificate guarantees firewall security.

ISVs' Liability Concerns

ISVs have a different liability concern, particularly when their software fails in the field. For example, suppose an SCL says that an ISV's software is "certified to not cause problem X." If the software causes problem X and fails, and the ISV faces legal problems, can the ISV use their SCL certificate as evidence of due diligence? Can the ISV assign blame to the SCL? The answer to the first question is "probably," and the answer to the second question depends on what "certified to not cause problem X" means. If the certification was based on objective criteria and the process was performed properly, the ISV probably cannot blame the SCL. If the process was improperly applied, the SCL may be culpable. If subjective criteria were applied, the answer is unclear.

If the SCL used consensus building to develop their certification process, the question that may someday be tested in the courts is whether abiding by an industry consensus on reasonable criteria protects SCLs from punitive damages. Generally speaking, as long as a professional adheres to defined standards, punitive damages are not administered. Professions such as medicine, engineering, aviation, and accounting have defined standards for professional conduct.

Lack of Professional Standards

Software engineering has never had such standards, although several unsuccessful attempts to do so have been staged. Also, there are no state-of-the-practice rules to determine if code meets professional standards. For example, the title "software engineer" is legally invalid in 48 of 50 states. In these states, the title "engineer" is reserved for people who have passed state-sanctioned certification examinations to become professional engineers [1]. Because the software engineering field does not have professional standards, it could also be argued that the actions of organizations such as the ICSA are laudable.

Because software engineering has no professional organization to accredit its developers, the approach taken by the ICSA could also be argued in a court of law to be state of the practice. If argued successfully, software developers whose software passed the certification process could expect to avoid punitive damages. But if these state-of-the-practice standards are deliberately weak, even though consensual, satisfaction of the standards may fail to persuade a jury. It is widely held by the public that the policy of industry self-regulation has failed. When those being forced to comply are those making the rules, are the rules trustworthy? Challenges in the courts could be foreseen, claiming a conflict of interest. This would invalidate claims that consensus-based standards sufficiently protect customers.

However, the commercial aviation industry is an example of the successful application of industry-guided standards. Rigorous software guidelines in the DO-178B standard were approved through industry and government consensus. These software safety guidelines remain the most stringent software certification standards in the world. It is clear that the Federal Aviation Administration's influence played a role during the formation of these standards.

There are self-correcting mechanisms that work to some degree in self-policing industries. If an industry such as air travel failed to police itself, it would lose so much favor with its customer base that the entire industry could fail.

Limiting ISV Liability

Possibly the best defense for any ISV is the use of disclaimers, not reliance on an SCL. There is a perverse advantage to disclaiming one's own product. The less competent an ISV portrays itself to be, the lower the standard of professionalism to which it will be held. Taking this principle to an extreme, we might suggest that a disclaimer be included in a comment at the top of each program stating, "This software was developed by incompetent people trying to learn how to program, and it probably does not work." The degree to which this tongue-in-cheek disclaimer reflects reality is a sad commentary on the state of our industry. But until more

cases are tested in the courts, no one knows how much protection software disclaimers will afford.

Article 2B

There is one more interesting development that has occurred: a draft of Article 2B of the Uniform Commercial Code (UCC) (which pertains to computers and computer services) was released Nov. 1, 1997 [2]. Article 2B will play an important role in defining software warranties. Article 2B will only serve as a model template, and each state in the United States will be responsible to modify it to their standards before adopting it as law. Further, Article 2B has the potential to relax the liability concerns that might force an ISV to use a certification laboratory. This could turn out to be a disaster for those parties most concerned with software quality.

Conclusion

Before we can determine what role SCLs will play in software liability, we must wait for more cases to be tested in court to see to what standard of professionalism ISVs are held. If the criteria for which SCLs test are not meaningful, SCLs will find that neither developers nor consumers of software care about the certification process.

For SCLs to succeed, it also is imperative that they employ accurate assessment technologies for objective criteria. If SCLs do this, malpractice suits against them will be difficult to win unless they mishandle a particular case or make false statements.

This article is entitled "Software Certification Laboratories: To Be or Not to Be Liable" because until these hard issues are resolved, it is hard to measure the degree of liability protection afforded an ISV by hiring the services of an SCL. Nonetheless, if SCLs can measure valuable criteria (and by this I do not mean "lines of code") in a quick and inexpensive manner, SCLs have the ability to foster greater software commerce between vendors and consumers. This could move an SCL certificate from being viewed as a tax to a trophy. ◆

About the Author

Jeffrey Voas is a co-founder of and chief scientist for Reliable Software Technologies and is currently the principal investigator on research initiatives for the Defense Advanced Research Projects Agency and the National Institute of Standards and Technology. He has published over 85 refereed journal and conference papers. He co-wrote *Software Assessment: Reliability, Safety, Testability* (John Wiley & Sons, 1995) and *Software Fault-Injection: Inoculating Programs Against Errors* (John Wiley & Sons, 1997). His current research interests include information security metrics, software dependability metrics, software liability and certification, software safety and testing, and information warfare tactics. He is a member of the Institute of Electrical and Electronics Engineers and he holds a doctorate in computer science from the College of William & Mary.

Reliable Software Technologies
21515 Ridgetop Circle, Suite 250
Sterling, VA 20166
Voice: 703-404-9293
Fax: 703-404-9295
E-mail: jmvoas@rstcorp.com

References

1. Jones, C., "Legal Status of Software Engineering," *IEEE Computer*, May 1995.
2. UCC Article 2B (Draft), November 1997, the American Law Institute and the National Conference of Commissioners on Uniform State Laws.

CMM, from page 20

time and within budget. Best of all, we achieved exceptional customer satisfaction, which is, after all, what counts. ◆

About the Author

Rita Hadden is the information systems performance practice leader at Project Performance Corporation. She has provided leadership, coordination, and coaching on more than 70 software projects for more than 35 organizations. Her software engineering and management experience includes 28 years working with multiple teams of developers and managers. She has successfully managed cross-platform information system projects for the private and public sectors. She is an acknowledged leader in industry best practices, software process improvement, and corporate culture change. She has helped organizations worldwide mature their software capabilities and meet their business objectives. She is certified by the SEI as a lead software process assessor.

Project Performance Corporation
20251 Century Boulevard
Germantown, MD 20874
Voice: 301-601-1810
E-mail: rhadden@ppc.com.

Engineering Practices for Statistical Testing

Jesse H. Poore, *University of Tennessee*
Carmen J. Trammell, *Software Engineering Technology, Inc.*

This article describes the application of statistical science to the testing and evaluation of software and software-intensive systems. Engineering practices are described for statistical testing based on a usage model, which is an engineering formalism that represents the use of a system in a specific environment or situation, or for a specific customer class. Engineering practices for statistical testing are based on a view of software use as a stochastic process and of software testing as a problem amenable to statistical solution.

When a population is too large for exhaustive study, as is the case for all possible uses of a software system, a statistically correct sample must be drawn as a basis for inferences about the population. In statistical testing of software, testing is treated as an engineering problem to be solved by statistical methods. Figure 1 shows the parallel between a classical statistical design and statistical software testing.

Under a statistical protocol, the environment of use can be modeled, and a statistically valid statement can be made about the expected operational performance of the software based on its test performance.

Statistical testing refers to the application of statistical science to testing software-intensive systems. It begins with characterizing all possible scenarios of use, includes analytical advice on design for testability, and ends with random testing to support estimates of the reliability of the system in field use.

A usage model is a characterization of all possible scenarios of software use at a given level of abstraction. Usage models can be constructed before code is written, and the model-building process can lead to improvements in the software specification that enhance testability.

A test case is any traversal of the usage model. A random test case is a traversal of the usage model based on state transitions that are randomly selected from a usage probability distribution.

Certification means attaining reliability and confidence goals for an environment of use following a protocol of demonstration. This protocol must be well defined, open to evaluation, and repeatable. As with any costly testing program, it is important that low-quality software be rejected (for example, when selecting commercial-off-the-shelf software), or returned to development (for revision and verification) quickly and inexpensively, and that testing continue only if progress toward certification of the software is being made.

Engineering practice refers to procedures that can be applied to problems of a recognizable type to achieve predictable and repeatable results. Engineering practices are derived from an appropriate science base, but the theoretical details of the science are organized, packaged, and often automated to be unobtrusive during application. Engineering practices are designed to get work done rapidly and correctly. The statistical testing process involves the six steps depicted in Figure 2. The engineering practices for each step are described in the succeeding sections.

Operational Usage Modeling

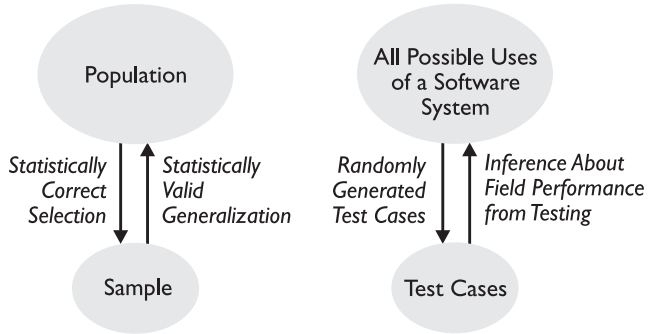
Building Model Structure

Usage models characterize the infinite population of scenarios of use. Usage models are built from specifications, user guides, or even existing systems. The user might be a human, a hardware device, another software system, or some combination. More than one model might be constructed for a single system if there is more than one environment of interest.

The basic task in model building is to identify the states of use of the system and the possible transitions among states of use. This information is encoded into highly structured Markov chains [1] in the form of directed graphs and stochastic matrices. Every possible scenario of use, at the chosen level of abstraction, is represented by the model. Thus, every possible scenario of use is represented in the analysis, traceable on the model, and potentially generated from the model as a test case. Figure 3 portrays a simple usage model as a directed graph with transition probabilities on the arcs.

Models should be designed in a standard form that consists of connected submodels with a single entry and single exit. States and arcs can be expanded like macros. Submodels

Figure 1. *Parallel between a statistical design and software testing.*



of canonical form can be collapsed to states or arcs. This permits model validation, specification analysis, test planning, and test case generation to occur on various levels of abstraction. The structure of the usage models should be reviewed with the specification writers, the real or prospective users, the developers, and the testers. Users and specification writers are essential to represent the application domain and the workflow of the application. Developers get an early opportunity to see how the system will be used and look ahead to implementation strategies that take account of use and work-flow. Testers are typically the usage model designers and therefore get an early opportunity to plan certification and to define and automate the test environment.

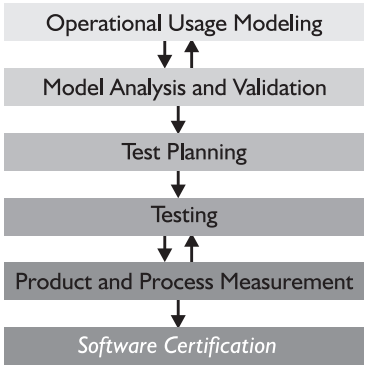
Most usage modeling experience to date is with embedded real-time systems, application program interfaces, and graphical user interfaces. Models as small as 20 states and 100 arcs have proven highly useful. Typical models are on the order of 500 states and 2,000 arcs; large models of more than 2,000 states and 20,000 arcs are in use. Even the largest models developed to date are small in comparison to similar mathematical models used in other fields of science and engineering and are manageable with available tool support.

Assigning Transition Probabilities

Transition probabilities among states in a usage model come from historical or projected usage data for the application. Because transition probabilities represent classes of users, environments of use, or special usage situations, there may be several sets of probabilities for a single model structure. Moreover, as the system progresses through the lifecycle, the probability set may change several times based on maturation of system use and availability of more information.

When extensive field data for similar or predecessor systems exists, a probability value may be known for every arc of the model. For new systems, one might stipulate expected practice based on user interviews, user guides, and training programs. This is a reasonable starting point but should be open to revision as new information becomes available.

Figure 2. *The statistical testing process.*



Generating Transition Probabilities

An alternative to the direct assignment of transition probabilities just discussed is to generate them as the solution to a system of equations [2]. Usage models can be represented by a system of constraints (written as equations or inequalities in terms of the transition probabilities as variables). The matrix of transition probabilities can be generated as the solution to the system. In general, three forms of constraints are used to define a model:

- Structural constraints are so named because they define model structure: the states themselves and both possible and impossible transitions among the usage states.
- Usage constraints represent information about known or expected patterns of system use.
- Management constraints reflect controls on the testing process to enforce budget, schedule, or policy decisions.

Probability values can be related to each other by a function to represent what is known about the relationship without overstating the data and knowledge. Most usage models can be defined with extremely simple constraints.

Engineering Practice for Operational Usage Modeling

Step 1. Identify the system boundary and all hardware, software, and human users of the software and the stimuli they can send the software.

Step 2. Define the structure of the usage model in terms of the possible sequencing of stimuli. Identify any areas where the software specification will result in excessive (as opposed to essential) complexity and cost in system development. Make recommendations for possible simplification.

Step 3. Define the important environments of use for the software, e.g., routine use, hazardous use, malicious use, maximum capacity use, and determine the number of environments to be modeled. Continue the process for each model.

Step 4. Define the transition probabilities of the usage model.

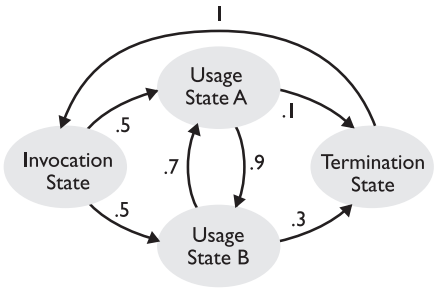


Figure 3. *Markov chain usage model.*

Model Analysis and Validation

Long-Run Characteristics of the Usage Model

A Markov chain is a thoroughly studied mathematical model for which a standard set of statistics exists. In this case, the standard statistics calculated for the usage model have important interpretations for resource allocation, safety analysis, test planning, and field support. Statistics that are routinely calculated from the model and used for these purposes include the following:

Long-run occupancy of each state – the usage profile as a percentage of time spent in each state.

Occurrence probability – probability of occurrence of each state in a random use of the software.

Occurrence frequency – expected number of occurrences of each state in a random use of the software.

First occurrence – for each state, the expected number of uses of the software before it will first occur.

Expected sequence length – the expected number of state transitions in a random use of the software; the average length of a use case or test case.

Analytical results are studied during model validation, and surprises are not uncommon. Parts of systems thought to be unimportant might get surprisingly heavy use, while parts that consume a large amount of the development budget might see little use. Since a usage model is based on the software specification rather than the code, it can be done early in the lifecycle to inform the development process as well as testing and certification.

Engineering Practice for Model Analysis and Validation

Step 1. Generate the standard analytical results for the model. Interpret analytical values in terms of the specification and expected usage to validate their correctness or reasonableness.

Step 2. Change the model structure or constraints if necessary. Changes to the structure may be needed to correctly represent the specification; changes to constraints may be needed to correctly represent usage or test management issues.

Step 3. If the model has been changed, repeat Steps 1 and 2.

Step 4. Generate some test cases and confirm that they look realistic; if not, return to Step 2.

Step 5. Use the model and its implications to inform development activities such as performance planning, correctness verification, safety analysis, and test planning.

Test Planning

Crafted (Nonrandom) Test Cases

There are compelling reasons for creating special, nonrandom test cases. Such testing can remove uncertainty about how the system will perform in various circumstances and can contribute to effectiveness and control over all testing, both crafted and random.

Following are types of nonrandom testing that may be useful prior to random testing.

Model coverage tests. Using just the structure of the model, a graph-theoretic algorithm generates the minimal sequence of test events (least cost sequence) to cover all arcs (and therefore all states). If it is practical to conduct this test, it is a good first step in that it will confirm that the testers know how to conduct testing and evaluate the results for every state of use and every possible transition.

Mandatory tests. Any specific test sequences that are required on contractual, policy, moral, or ethical grounds can be mapped onto the model and run.

(Nonrandom) regression tests. Regression test suites can be mapped to the model. This is an effective way to discover the redundancy in the test suite and assess its omissions.

Critical but unlikely use. Critical states, transitions, and subpaths that would have low likelihood of arising in a random sample can be identified from the model and tested by crafted cases.

Importance tests. Importance sampling can be implemented by adding management constraints and an objective function that will produce the transition probabilities that will emphasize the "value" in the sampling process.

Partition testing. The usage model can be used to identify and define partitions to gain sampling efficiency.

Random Test Cases

Random test cases may be automatically generated from the usage model. Each test case is a "random walk" through the model, from the initial state to the terminal state. Test cases may be generated as scripts for human testers or as input sequences for automated testing. Post-processing of test cases often further facilitates human or automated evaluation. One may generate as large a set of test cases as the budget and schedule will bear and establish bounds on test outcomes before incurring the cost of performing the tests.

Engineering Practice for Test Planning

Random testing should begin only after all crafted testing has been completed.

Step 1. Using the expected test case length derived during model analysis, estimate and generate the number of random test cases that can be run within the schedule and budget.

Step 2. Define the best-case scenario. Assume that no failures occur in random testing, and determine the values of product quality and process sufficiency that can be achieved by running the number of test cases generated in Step 1. (These measures are described in the "Product and Process Measurement" section.)

Step 3. Define the worst-case scenario. Assume some profile of failures and construct a failure log based on the profile. (This may be done for several scenarios.) Again, determine the values of product and process certification measures under the scenario. The comparison of these values with actual certification goals will reveal how much bad news the budget and schedule can absorb.

Step 4. Analyze the coverage of model states, arcs, and paths that will occur.

Step 5. Analysis might show that testing as planned and budgeted cannot, even in the case of no failures, satisfy requirements for model coverage or demonstrable reliability. Given this fact, one may choose to either revise goals or revise plans.

Testing

It is essential to the integrity of the certification process to maintain experimental control throughout random testing. Experimental control refers to complying with the assumptions associated with a statistical protocol. Results must be evaluated consistently. The team must ensure a common understanding of all test materials and policies so that consistent test decisions are

made. Other steps to ensure experimental control are given in [3, Chap. 17].

Engineering Practice for Testing

Step 1. Hold the specification and oracle constant for each version of the software that is tested.

Step 2. Sustain the conditions in the environment throughout testing.

Step 3. Monitor the performance of human testers to prevent "drift."

Step 4. Run test cases in the order in which they are generated.

Step 5. Schedule regular communication among testers for discussion of matters that may affect test judgment.

Step 6. Log all failures.

Step 7. Maintain at least two testing chains, i.e., testing records encoded as Markov chains, one for the current version of the software and one for the history of testing across all versions. The current-version testing chain will be used for certification and stopping decisions. The historical testing chain will be used to study the development and testing processes.

Step 8. If one or more failures occur during testing of the current version, a decision must be made about whether to stop testing. Many factors may be involved, including the nature of the failures, schedules, and organizational policies. Monitor reliability, confidence, and convergence of the testing chain to the usage model to guide stopping decisions.

Step 9. If no failures are seen during testing, base stopping decisions on reliability, confidence, and convergence measures together with remaining schedules and budgets.

Product and Process Measurement

The usage model from which the test cases are generated is called the "usage chain." A chain of initially identical structure is developed to record actual testing experience, called the "testing chain." The progress of testing is monitored by tracking measures calculated from these two chains.

Product Measures

A reliability measure is calculated from the testing chain along with confidence intervals. This reliability is defined strictly in terms of the failure experience recorded in the testing chain; there are no other mathematical assumptions. This definition of reliability is applicable whenever testing has revealed one or more failures. (When testing reveals no failures, distributional models should be used, e.g., [3, Chap. 5; 4].)

Process Measures

An information theoretic comparison of the usage and testing chains is computed to assess the degree to which the testing experience has become representative of expected field use. Its graph will have a terrace-like appearance of declines and plateaus. The trend in the measure reveals the rate at which the usage and testing chains are becoming indistinguishable. As the two converge, it becomes less likely that new information will be gained by further testing.

Certification

The certification process involves ongoing evaluation of the merits of continued testing. Stopping criteria are based on reliability, confidence, and uncertainty remaining. Decisions to continue testing are based on an assessment that the goals of testing can still be realized within the schedule and budget remaining.

In most cases, users of statistical testing methods release a version of the software in which no failures have been observed. Reliability estimates such as those in [3, Chap. 5; 4] are recommended in this case.

Software is sometimes released with known faults. If the test data includes failures, reliability and confidence may be calculated from the testing chain. The reliability measure computed in this manner reflects all aspects of the sequences tested, including the probability weighting defined by the usage model.

Certification is always relative to a protocol, and the protocol includes the entire testing process and all work products. An independent audit of testing must be possible to confirm correctness of reports. An independent repetition of the protocol should produce the same conclusions to within acceptable statistical variation.

Conclusions

The model construction and validation process is an investment in understanding how the system will be used. Several calculations flow directly from the usage models without further assumptions that quantify the size and complexity of the testing problem. For many practitioners, this quantitative, defensible characterization of size and complexity provides insights that are overwhelming. Practitioners have variously slashed requirements, pruned user options, erected firewalls, extended schedules, extended budgets, initiated test automation efforts, and decimated reliability goals. If one believes that the usage model accurately captures the capability described in the specifications and that the probabilities represent the intended environment of use, the calculations and conclusions based on them are inescapable.

Work in progress is focused on composition of usage models from components, synthesizing information about the whole from the components, and combining testing information across the product lifecycle. ◆

About the Authors

Jesse H. Poore is professor of computer science at the University of Tennessee. He works with industrial and government sponsors on practical software engineering problems. He served as assistant to the president for information technology and professor of computer science at the Georgia Institute of Technology. In 1983, he was executive director of the Committee on Science and Technology in the U.S. House of Representatives. From 1971 to 1980, he was associate professor of mathematics and director of the Computing Center at Florida State University. He holds a doctorate in information and computer science from Georgia Tech.

Department of Computer Science
University of Tennessee, 107 Ayres Hall

Knoxville, TN 37996-1301
Voice: 423-974-5784
Fax: 423-974-4404
E-mail: poore@cs.utk.edu

Carmen J. Trammell is a software consultant with Software Engineering Technology, Inc. She was formerly research assistant professor and manager of the Software Quality Research Laboratory in the Department of Computer Science at the University of Tennessee. She has held technical and management positions in software projects through Oak Ridge National Laboratory, Martin Marietta Energy Systems, and Software Engineering Technology and currently works with industry and government on software engineering process definition and improvement. She has done software development and testing for the U.S. Army and the U.S. Navy, academic teaching on military bases overseas, industrial training for Department of Defense (DoD) contractors, and research and development under contract to the DoD Software Technology for Adaptable, Reliable Systems Program. She has a master's degree in computer science and holds a doctorate in psychology from the University of Tennessee.

Software Engineering Technology, Inc.
5516 Lonas Road, Suite 110
Knoxville, TN 37909
Voice: 423-450-5151 ext. 240
Fax: 423-450-9110
E-mail: trammell@toolset.com
Internet: http://www.toolset.com

References

1. Kemeny, J.G. and J.L. Snell, *Finite Markov Chains*, Van Nostrand, Princeton, N.J., 1960.
2. Walton, G.H., *Generating Transition Probabilities for Markov Chain Usage Models*, Department of Computer Science, University of Tennessee, Knoxville, Tenn., 1995.
3. Poore, J.H. and C.J. Trammell, *Cleanroom Software Engineering: A Reader*, Oxford, England, Blackwell Publishers, 1996.
4. Miller, K.W., et al., "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Transactions on Software Engineering*, January 1992.

Coming Events

Tenth Annual Software Technology Conference (STC '98)

Dates: April 19-23, 1998
Location: Salt Palace Convention Center, Salt Lake City, Utah
Co-hosts: Ogden Air Logistics Center commander and the Software Technology Support Center
Contact: Dana Dovenbarger
Voice: 801-775-4932 DSN 777-7411
E-mail: dovenbar@oodis01.hill.af.mil

IEEE Computer Society International Conference on Computer Languages 1998

Dates: May 14-16, 1998
Location: Loyola University Chicago, Chicago, Ill.
Sponsors: IEEE Computer Society Technical Committee on Computer Languages in cooperation with the Association for Computing Machinery Special Interest Group on Programming Languages
Contact: Internet: http://www.math.luc.edu/iccl98/

7th IEEE North Atlantic Test Workshop

Dates: May 28-29, 1998
Location: West Greenwich, R.I.
Subject: Provides a forum for discussions on the latest issues relating to higher quality, more economical, and more efficient testing methods and designs. The workshop will focus on "Reliability and Testing Issues for the 21st Century."
Sponsors: IEEE Computer Society, Test Technology Technical Committee, and the University of Rhode Island
Contact: Jim Monzel
Voice: 802-769-6428
Fax: 802-769-7509
E-mail: jmonzel@vnet.ibm.com

Quality Week '98, Conference "Countdown to 2000"

Dates: May 26-29, 1998
Location: San Francisco, Calif.
Subject: Focuses on advances in software test technology, quality control, risk management, software safety, and test automation, software analysis methodologies, and advanced software quality processes.
Contact: Rita Bral
Voice: 800-942-SOFT (800-942-7638)
Fax: 415-957-0730
E-mail: qw@soft.com
Internet: http://www.soft.com/QualWeek/QW98

Conference on Risk Management Software Engineering Institute (SEI)

Dates: June 8-10, 1998
Location: Crystal Gateway Marriott, Crystal City, Va.
Sponsor: Software Engineering Institute
Contact: SEI Customer Relations
Voice: 412-268-5800
Fax: 412-268-5758
E-mail: customer-relations@sei.cmu.edu

Engineering Disasters . . .

Norman F. Simenson
Federal Aviation Administration

Engineering disasters are always *the result of bad management and* never *the result of bad engineering—or almost always. This article describes three lines of defense any manager can use to guard against bad engineering.*

Any experienced program manager (PM) will quickly recognize that I have indulged in a little hyperbole in the following article to emphasize a number of deeply held beliefs.

I stress that the article is *not* intended to provide a shield for those PMs who go to extreme lengths to avoid any risks. By definition, good PMs are risk takers. That is their strength. Risk taking becomes a weakness only if the PM does not understand the difference between gambling and controlled risk taking. In real life, the knowledgeable risk taker will (almost) always outperform the gambler *and* the ultraconservative—by a wide margin. A word to the wise: Most insurance companies are highly profitable—by design, not by accident.

If I seem to have let engineers off easily, that is not the intent. Engineers are fully responsible for the specific task assigned to them and must answer in full for a competent job. However, only the PM and chief engineer are responsible for the full job. They must shoulder the full blame if the project fails. The point is, if the project is lost for "want of a nail[1] ...," for the failure of one—or even several— lower-level engineers, or because of a few minor glitches, the program manager and chief engineer are mightily to blame.

Bad Management

Engineering *development* disasters are *always* due to bad management and *never* to bad engineering because, for any but the smallest noncritical project, the PM must do risk management. The plea "failed due to uncontrollable forces" or "my team let me down" is almost invariably an admission of bad management. Worse, it means not only that the PM was inept but also does not have a clue about how to improve. A manager is not merely an overpaid administrator who handles the budget and doles out the work. A manager is the principal defense against the real-life disasters that can destroy any program.

Simply put, the PM must expect and plan for *all* engineering problems. This includes workable contingency plans that will prevent problems from impacting schedule and cost. Even with the best of engineers, creativity cannot be planned for and scheduled like a train timetable; some slippage is bound to occur. The most mundane engineering projects contain requirements for a substantial degree of good engineering and creativity. Provision must be made for things not going as planned; otherwise, who needs a PM?

Contingency Plans

There are all sorts of contingency plans and safeguards against problems due to bad engineering. Some will also work for almost any unexpected technical difficulties. The different problem categories should be treated differently but have some overlap and similarities for planning purposes. Here, we will only take a superficial view of planning to prevent the impact of bad engineering.

Assessing Teammates Strengths

The first line of defense for the PM is to correctly assess the abilities of the engineer, then correctly assign the engineer to a job. The "hallway" technique of assigning people is a sure recipe for disaster. This technique assumes that everyone within the same labor category is equivalent, so it must be okay to assign the first warm body of the appropriate category to pass the manager's door in the hallway to the job at hand. The amazing thing is not how often the warm body fails but rather, how often they succeed.

For any given job, there are four types of people. One type has never done a similar job and may possess some or most of the necessary skills but is fundamentally an unknown quantity. The other three types have done a similar job before, but one has failed at it, one has performed acceptably, and one has performed outstandingly. Clearly, depending on the skill pool and if other priorities permit, the choice is one of the latter two types. If the task is sufficiently critical, the choice may be only someone of the last type. The PM must give special care to the selection of the chief engineer and plan to accommodate their strengths and weaknesses. (A good chief engineer will devise a similar plan to accommodate the PM.)

If people of the first two types only are available, the manager must spend considerably more effort to decide among the candidates. But if managers lack qualified people for a program, they must have the courage to tell their management and insist on a workable solution. Agreeing to take on an impossible program is managerial malpractice. It is impossible to categorize people with respect to a job if the manager has no notion of what skills the proposed job requires or what skills the candidate's previous job required. Both are easily determined, as well as the performance of engineers on past jobs, but not without effort on the manager's part.

In deciding to give someone "another chance," the manager must assess whether that someone has demonstrated an unremediated lack of some

necessary technical or personality skill. To determine if someone deserves a crack at something that will probably stretch their abilities, the manager must decide whether that person has the basic technical and personality skills needed and is likely to rise to the occasion. In either case, it must be assumed that an individual that has failed at or has never performed a task will need extra support in the form of training, mentoring, and supervision. That person is also likely to take longer to perform the task than one of the successful people. Due allowance must be made.

Provide Support and Resources

The second line of defense is to ensure that the engineer assigned to the task has the necessary support and resources. If these are not supplied in sufficient quality and quantity, the correlation between past and future success is bound to be poor. No matter how skillful the carpenter, a good job of nailing one board to another is unlikely without a hammer or nails or if one board is at another site two miles away. Unfortunately, it is common to overload an outstanding performer. It is easy to assume that a fraction of one good worker is better than all of a weak one or that a strong performer can make do with significantly less, i.e., inadequate support and significantly fewer resources. Almost equally disastrous is the practice of rationing resources "impartially" with little or no regard to the difficulty of the job or the ability of the performer. It does not work for parents, and it will not work for managers.

Plan Redundancy

A third line of defense is planned redundancy. It always amazes me how top managers will skimp on a program when sufficient planning and use of resources will ensure delivery, then spend resources, seemingly without limit, once failure is imminent and generally unavoidable. If you plan for failure upfront, it is avoidable. Suppose the only candidates for a job are of the first two types of employee described earlier. In this case, it is best to start two or even three on different aspects of the same job in parallel. When one or two of your candidates fail, go with the successful candidate. Never make the mistake of shifting resources from a succeeding candidate to a failing candidate. This is a formula to ensure that everyone will fail, which has been proven repeatedly in military combat. Cut your losses quickly—this includes getting rid of bad engineers quickly. (Bad engineers are those who are not only incompetent but also take no responsibility for, and therefore do not learn from, their mistakes.)

If you wind up with all failing candidates, make sure you have the machinery in place to detect this early and, as each candidate fails, switch to an alternative strategy. This may involve supporting or replacing them with a more successful performer. This is one reason why you want to plan to support the weak candidate with more time and resources and with more supervision at the outset. It also is a reason never to start a program without some engineering reserve (which may be no more than potential overtime). Programs that start with everyone already overcommitted always fail. For critical tasks, it is a good idea to backup even the best engineers. Consider this "bus" insurance. (You must always worry about and plan for your key engineers being hit by a bus or other catastrophe.)

Using a low-risk, redundant approach assures against the failure of any high-risk approach. The resources expended on the redundant alternative(s) should be considered an insurance premium. Where a parallel approach strategy is used, multiple successes will speed the result, so not much is lost if properly planned.

There are many other strategies a competent manager can use to ensure against bad engineering or other disasters. Enough strategies should be used to reduce the risk of failure to a tolerable level at an acceptable cost.

Conclusion

In closing, I make three points. First, nothing in this article applies particularly to software managers. Second, through the software process improvement method and related techniques initiated by the Software Engineering Institute and others, software has done more than its fair share to raise "scientific" management from a simple collection of heuristics toward the status of a science. Third, despite the general wailing, breast-beating, and gnashing of teeth, software must be doing something right. There has been nothing like the headlong rush to software since the similar rush to electronics after World War I. The average automobile of today has more software in it than the first Apollo spacecraft to arrive at the moon less than 30 years ago! ◆

About the Author

Norman F. Simenson is vying for the title of "grand old man" of hard real-time software. (He has an IBM Engineering School diploma dated March 1955 and signed by T.J. Watson—Sr. and Jr.) He also has various advanced degrees from the University of Michigan.

Currently, he is a member of a Software Engineering Group at the Federal Aviation Administration (FAA). This group is charged with the responsibility to improve the FAA's software engineering processes and practices. He participates in most Software Capability Evaluations performed at the FAA and also is editor (and resident curmudgeon) of the *FAA Software Engineering Process Group Interface* newsletter: www.faa.gov/ait/sepg/news1.htm.

Simenson has been a programmer, program manager, chief engineer, chief architect, and chief scientist. He has worked with real-time systems for about eight years.

DOT/FAA/AIT-5
800 Independence Avenue, S.W.
Washington, D.C. 20591
Voice: 202-267-7431
Fax: 202-267-5080
E-mail: norm_simenson@faa.dot.gov

Note

1. "For want of a nail, a shoe was lost. For want of a shoe, a horse was lost. For want of a horse, a message was lost. For want of a message, a battle was lost. For want of a battle, a war was lost. For want of a war, a kingdom was lost. And all for the want of a nail!"

CROSSTALK

Sponsor	***Lt. Col. Joe Jarzombek*** 801-777-2435 DSN 777-2435 jarzombj@software.hill.af.mil
Publisher	***Reuel S. Alder*** 801-777-2550 DSN 777-2550 alderr@software.hill.af.mil
Managing Editor	***Forrest Brown*** 801-777-9239 DSN 777-9239 brownf@software.hill.af.mil
Senior Editor	***Sandi Gaskin*** 801-777-9722 DSN 777-9722 gaskins@software.hill.af.mil
Graphics and Design	***Kent Hepworth*** 801-775-5555 ext. 3027 hepwortk@software.hill.af.mil
Associate Editor	***Lorin J. May*** 801-775-5555 ext. 3026 mayl@software.hill.af.mil
Editorial Assistant	***Bonnie May*** 801-775-5555 ext. 3023 mayb@software.hill.af.mil
Features Coordinator	***Heather Winward*** 801-775-5555 ext. 3028 winwardh@software.hill.af.mil
Customer Service	801-777-8045 DSN 777-8045 custserv@software.hill.af.mil
Fax	801-777-8069 DSN 777-8069
STSC On-Line	http://www.stsc.hill.af.mil
CROSSTALK On-Line	http://www.stsc.hill.af.mil/ Crosstalk/crostalk.html
ESIP On-Line	http://www.esip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:

Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

E-mail: custserv@software.hill.af.mil
Voice: 801-777-8045, DSN 777-8045
Fax: 801-777-8069, DSN 777-8069

Editorial Matters: Correspondence concerning *Letters to the Editor* or other editorial matters should be sent to the same address listed above to the attention of *CROSSTALK Editor* or send directly to the senior editor via the E-mail address also listed above.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the *CROSSTALK* editorial board prior to publication. Please follow the *Guidelines for CROSSTALK Authors*, available upon request. We do not pay for submissions. Articles published in *CROSSTALK* remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Most material in *CROSSTALK* may be reprinted at no charge. Coordinate reprint requests with *CROSSTALK*.

Trademarks and Endorsements: All product names referenced in this issue are trademarks of their companies. The mention of a product or business in *CROSSTALK* does not constitute an endorsement by the Software Technology Support Center (STSC), the Department of Defense, or any other government agency. The opinions expressed represent the viewpoints of the authors and are not necessarily those of the Department of Defense.

Coming Events: We often list conferences, seminars, symposiums, etc., that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the *CROSSTALK* Editorial Department.

STSC On-Line Services: STSC On-Line Services can be reached on the Internet. World Wide Web access is at http://www.stsc.hill.af.mil. The STSC maintains a Gopher server at gopher://gopher.stsc.hill.af.mil/. Its ftp site may be reached at ftp://ftp.stsc.hill.af.mil. The Lynx browser or gopher server can also be reached using telnet at bbs.stsc.hill.af.mil or by modem at 801-774-6509 or DSN 775-3602. Call 801-777-7989 or DSN 777-7989 for assistance, or E-mail to portr@software.hill.af.mil.

Publications Available: The STSC provides various publications at no charge to the defense software community. Fill out the Request for STSC Services card in the center of this issue and mail or fax it to us. If the card is missing, call Customer Service at the numbers shown above, and we will send you a form or take your request by phone. The STSC sometimes has extra paper copies of back issues of *CROSSTALK* free of charge. If you would like a copy of the printed edition of this or another issue of *CROSSTALK*, or would like to subscribe, please contact the customer service address listed above.

The ***Software Technology Support Center*** was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. *CROSSTALK* is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, UT 84056. *CROSSTALK* is distributed without charge to individuals actively involved in the defense software development process.

COTS Reliability: An Oxymoron?

I've spent the last hour trying to come up with an ironic opening line about the reliability of commercial software, but every time I try to type something my computer locks up. So instead I'll state my point outright: I support legitimate military efforts to integrate commercial-off-the-shelf (COTS) software, so long as the companies that created the software for my computer aren't allowed anywhere within the same time zone as a U.S. military base.

The companies that created the software for my computer are all led by former executives of "Yugo Corp." Their quality standards for cars ("Transmission is securely attached with *extra-strength* masking tape!") have translated over to their software reliability rates ("Crash per hour rate is below 2.5!"). For example, below are the results from a commercial translation program. The following sample E-mail message went from English to Portuguese and back to English.

English version: "Hey, what's up? I've been slamming out some of my hottest code ever, but our new chief bugs me. At first she seemed pretty on the ball, but she won't cut us any slack. I can't take it, so I'm laying low and hoping she gets downsized. Drop me a line some time, all right? Well, I've got to cruise to my pad and hit the sack. Take care, John."

Translated to Portuguese and back: "Hey, that is above? I have beaten for I am of some the my hottest code always, but of our new main errors me. Beginning in seemed consideravelmente in the sphere, but in it will not cut them flabby. I cannot make examination d, thus that I am placing point low e waiting I start downsized. He leaves me to fall a line some hour, all for the right? Well, I started to cross cushion and beat sack. Care of the taking, John."

To be fair, this was translated by a beta version of the program, which was probably created faster and cheaper than the government could form a steering committee to appoint a task force to schedule an investigation on which countries use Portuguese. The bad news is, down the road this task force would probably recommend buying the above software for all military correspondence in Portuguese, and judging by the translation quality, within a matter of weeks we would be at war with Brazil.

But since COTS appears to be the wave of the future, some companies may need practice building software according to stricter military standards. Before integrating their software into our systems, we should let commercial producers start by building military systems where quality problems won't impact our national security—such as creating missile guidance systems for Third-World military dictatorships:

Dictator: Is the missile ready for launch against the line-dancing, Big Mac-eating capitalist swine?
Minion: Yes, oh crazed despotic leader! Type in the launch code and press enter when you are ready.
Dictator (types code and presses enter): Arg! A little paper clip cartoon is asking if I want help!
Minion: Press "cancel" and hit "fire"! (Dictator presses "cancel" and "fire.")
Computer: ARE YOU SURE YOU WANT TO FIRE? <Yes> **<No>**
Dictator: Yes! Fire! (WHOOOOOOSH!!!!) Finally! Take that, you NATO ninnies!
Computer (pauses for several seconds)**:** GENERAL MISSILE DEFAULT. PRESS <Yes> TO RESTART.
Dictator: Hurry! Shut this rotten thing down and boot up our old system!
Computer: SHUTTING DOWN. PLEASE WAIT WHILE MISSILE RETURNS TO SILO.
Minion: Incoming!
Dictator: AAAAAAAAAAIIIIIIIIIIEEEEEEEEEEEEEEEEEEEEEEEEE!!!

As the military integrates more COTS software, there will be some adjustments. For example, we will need additional personnel to update our airplanes, tanks, missiles, and boats so that they can still communicate with each other whenever a COTS manufacturer decides to release a new version of software that has no perceivable advantage over its previous six upgrades (save for a cooler box). To maintain version control, perhaps we will someday see, for example, the F-16.3.11 Falcon fighter ("Now imports all leading spreadsheet formats!") or the M1.22B-Abrams tank ("New patch: turret now turns clockwise *and* counterclockwise!"). They would probably also need several crews to paint and repaint the extra version numbers on the equipment. I'm not sure whether I'm joking about all this or not.

With time and experience, I'm sure COTS software will settle into its proper, most logical uses. But as far as treating COTS as the end-all answer to our cost problems, I say, "Don't count your chickens before they hatch." Or as they say in Brazil, "He does not count its hens before they shock." — Lorin May

Got an idea for BACKTALK? Send an E-mail to mayl@software.hill.af.mil