Avoiding the Trial-by-Fire Approach to Security Incidents

NEWS AT SEI

Author

Moira West Brown

This article was originally published in News at SEI on: March 1, 1999

In previous issues of this column, we have written about the importance of considering security from product design or procurement through deployment and use. Being proactive about security is critical to mitigating your security risk. However, having good security measures in place won't prevent you from suffering computer security incidents, so it is also important to be prepared and proactive about detecting and responding to such incidents when they do arise: Organizations may be proactive about fire prevention, but it is still necessary to have detection (e.g., smoke detectors) and response (e.g., sprinklers and fire extinguishers) in place to address a possible fire. Moreover, fire drills are conducted to ensure that the planned response approach is effective and understood by all involved. An organization's approach to security must be just as comprehensive in its scope as fire prevention, detection, and response. This article will explore the range of options that exist in organizations today for detecting and responding to security incidents.

The trial-by-fire approach

Experience shows that most organizations don't think about how to respond to a computer security incident until after they have experienced a significant one! This problem is common; many organizations have not assessed the business risk of having no formal incident-detection and response mechanisms in place. More often than not, organizations receive reports informing them that they are involved in an incident from some other part--rather than identifying the incident themselves!

The problem stems from a lack of organizational recognition of the need for a comprehensive security infrastructure. It is not until after an ill-prepared organization has suffered a significant security incident that the real business risk and impact of such an event becomes apparent. There may be some organizational or management perception that network and host security is something that the system and network

administrators handle as a part of their day-to-day activities, or that security is handled by the organization's firewall. Sadly this perception is often incorrect on both counts. The priorities of such staff are primarily focused on maintaining basic support and operation of the vast amount of computing equipment in place. Firewalls may prevent some attacks, but can't prevent all attack types, and if not correctly configured and monitored they may still leave the organization open to a range of others. This approach, or lack of one, results in significant problems including:

- not knowing if or for how long a network or systems have been compromised
- not knowing what information is at risk, has been taken, or has been modified by intruders
- not understanding the methods that the perpetrator(s) use to gain access to systems
- not understanding what steps can be taken to stop the intrusion activity and secure the systems and network
- failure to identify in advance any possible adverse effect that steps taken in response to an incident may have on the company's ability to conduct business
- not knowing who has authority to make decisions related to containing the activity, contacting legal department/law enforcement, etc.
- delays in identifying and contacting the right people to inform them about the activity (both internally and externally)
- no recognized point of contact in the organization that external or internal parties know to report to

The volunteer approach

Some organizations have system and network administrators who are either interested or trained in computer security. Such individuals are better prepared to address security within their domain of authority--such as the machines in one department or operating unit, or the equipment on a given network segment. Within some organizations, various individuals may be working together to address security informally. This approach often stems from a group of individuals in the organization who see the need to address security even if the need is not recognized by higher level management. However, even then, having good people available doesn't mean that the organization is prepared to respond. Depending on the scope of the overall volunteer effort, it is likely that even with intrusion-detection software in place in parts or the organization, serious network security incidents may still go undetected. Although this approach is a marked improvement over the trial-by-fire approach, significant problems still remain including

- Serious intrusions may still go undetected.
- Volunteers may be able to deal with the technical issues, but may not understand or have the information available to assess the business consequences of any steps taken.
- Volunteers may not have the authority to apply the technical steps (e.g., disconnecting the organization from the Internet) or other actions they believe are necessary (e.g., reporting the activity to law enforcement or seeking the advice of

legal counsel).

- There may be delays in seeking and obtaining management approval to respond.
- Volunteers have no "bigger picture" of the overall detection and response activity.
- Volunteers may know in some cases who to contact internally, but anomalies may exist.
- Other individuals in the company who identify a possible security incident may not be aware of the informal group and may fail to report to it.
- An informal group is unlikely to have external recognition and support.

Company-supported approach

Regardless of the good intentions of technical experts or other staff members, the only effective approach to incident detection and response is to make it part of an organization-wide risk-management plan based on a foundation of management support at the highest level within the organization. Regardless of how such an approach is implemented, (whether by a geographically distributed or centrally located team, consisting of full- or part-time staff or supplemented with contract support), without management support, the effort will struggle to succeed. In addition to the foundation of management support, the empowered group must also be recognized internally and externally and prove its effectiveness, trustworthiness, and ability to all involved. Management authority and recognition are the foundation for success, but an effective detection and response service needs the trust and respect of the constituency served and others with whom the service will need to interact.

Teams established to address incident detection and response for organizations are known as computer security incident response teams (CSIRTs). Forming, staffing, and operating a CSIRT is not easy. However, if appropriately set up and empowered within an organization, a CSIRT can begin to gain the trust and respect necessary to address incident detection and response from a company-wide perspective. CSIRTs vary in structure, staffing, and the range of services provided based on the situation or need that they are trying to fulfill. Consider the need for a CSIRT in your own organization, whether it is company wide or just for your business unit or department. A recently published handbook, *Handbook for Computer Security Incident Response Teams (CSIRTs)*, is now available to help an organization determine the scope and range of services for a CSIRT and provide guidance in forming operational CSIRT policies and procedures.

Advocating a company-supported approach

Making the transition from a trial-by-fire or volunteer response effort to a company-supported one is not easy. The most important and often the most difficult challenge is convincing management of the business need for an effective and empowered CSIRT as part of an overall risk-management approach.

Waiting for a serious security incident to occur within your organization to convince management of the need is not a productive approach, nor will it necessarily be successful. Even after suffering a serious computer-security incident in which hundreds of systems are compromised, some organizations still don’t recognize the need for a formal incident-response capability. I remember one case in which I contacted a multi-national company to inform them of information that indicated that an intruder was gaining access to the company's corporate network through the Internet. As a result of the report, the company began to look at its systems and found that they had been seriously compromised for over six months. The company was able to identify many systems and internal networks that were compromised by the activity and the sensitive information available on those systems, but had no idea of the intruder’s motives nor the extent of the data that the intruder had copied or amended. A significant period of time elapsed and further compromises occurred before the organization established a

CSIRT.

Another organization that was compromised by an intrusion took the step of reinstalling all of its systems from known good backups--losing two weeks of production effort in the process--as they could not be certain what data might have been tampered with by the intruder. In this case, malicious modifications to the application under development could have resulted in loss of life if the application had failed during use. The organization involved promptly established a company-supported CSIRT.

One of the most important factors to document is the associated business risk or loss of any incident. This information must be presented in a form that will help management understand that the problem is a business one and not a technical one. I recall one case in which technical staff had great problems in gaining management attention regarding ongoing intrusions. It was not until the intrusion data was presented by describing the mission of each system in question rather than providing its hostname and operating-system version that management paid attention. Volunteers should attempt to document and present to management the impact of known intrusions and recorded losses.

The influence of insurance

I learned of one situation recently in which a security officer compromised the home system of a manager as a last resort to gain management recognition of the company's security risk. For the majority of us, such extreme measures are far too dangerous. In such cases, financial pressure from another source may be a last resort to gain management's attention. Pressure from insurance companies (seeking to limit exposure of losses resulting from network-security incidents) will provide a financial incentive for organizations to improve security measures in order to keep insurance premiums affordable.

In a recent insurance application in which I was involved, an insurance company requested information on what policies an organization had in place for virus prevention and control of defamatory or libelous information on public Web sites and mailing lists. Conspicuous by its absence were questions seeking an understanding of how well prepared the organization was to prevent, detect, and respond to computer security incidents--even if only from the perspective of preventing viruses or defamatory/libelous information being published on a public forum. It will not be long before insurance companies are asking the right questions in this area. In fact some already are, but their motives are slightly different. Just recently some insurance companies have begun to offer policies that provide organizations with financial protection for third-party damages resulting from network-security breaches. A prerequisite for such coverage is an associated network-security risk assessment. It is only a matter of time before insurance companies begin to request more information about network security and begin to raise the cost of general insurance coverage for companies that are ill prepared to detect and respond to computer-security incidents. Eventually, one way or another, organizations through trial by fire or financial incentives will realize the need for a CSIRT.

Be prepared

It is still not uncommon to find callers to the CERT® Coordination Center hotline who do not know what steps to take to report an incident within their own organizations. Although many callers know who their vendor is and maybe even who the organization's Internet service provider is, very few know to whom they should report a computer-security incident. Being prepared and knowing what to do in advance can help to further mitigate the damage from an incident. That is why it is very important that an organization advertise its CSIRT both internally and externally. As with emergency services, it is important to find out how to contact a CSIRT before it is needed in an emergency. It is also important to know in advance to whom the service can provide help and what information is needed to ensure that the CSIRT can provide the service requested.

To find out if your organization has a company-supported CSIRT, ask your security officer or system/network administrator, and consult your organization's security policies and practices. Some CSIRTs are members of the Forum of Incident Response

and Security Teams (FIRST). FIRST provides a list of its members and their contact information.

With millions of organizations now reliant on networks to conduct their businesses, it is a shocking fact that only a few hundred CSIRTs exist around the world today. Many of these CSIRTs continue to cite annual increases of 200% or 300% in the numbers of computer-security incidents reported to them and are struggling to keep pace with the number of incoming reports. Even with general improvements in the field of network security, a dramatic increase in the number of CSIRTs in existence today is urgently needed. More advocates are needed to help organizations understand the risks associated with the failure to detect and appropriately respond to computer-security incidents.

About the author

Moira J. West-Brown is a senior member of the technical staff within the CERT® Coordination Center, based at the SEI, where she leads a group responsible for facilitating and assisting the formation of new computer security incident response teams (CSIRTs) around the globe.

Before coming to the CERT®/CC in 1991, West-Brown had extensive experience in system administration, software development and user support/liaison, gained at a variety of companies ranging from academic institutions and industrial software consultancies to government-funded research programs. She is an active figure in the international CSIRT community and has developed a variety of tutorial and workshop materials focusing mainly on operational and collaborative CSIRT issues. She was elected to the Forum of Incident Response and Security Teams Steering Committee in 1995 and is currently the Steering Committee Chair. She holds a first-class bachelor's degree in computational science from the University of Hull, UK.

For more information

Contact Us

info@sei.cmu.edu

412-268-5800

Meeting the Challenges of Requirements Engineering

NEWS AT SEI

This article was originally published in News at SEI on: March 1, 1999

Software developers must consider requirements early in the development process--and consider them carefully. Errors that occur early in the software development life cycle, but are not discovered until late in the life cycle, are the costliest to fix. This realization has encouraged software engineers to focus increased attention on requirements, as a way to find improvements that will have a large impact.

While the Software Engineering Institute does not concentrate specifically on requirements engineering, all SEI work in software engineering management and technical practices bears some relation to requirements elicitation, analysis, or management. For example:

- Architecture analysis and model-based verification expose problems with early versions of program or system requirements.
- Developers must have significant flexibility on requirements if they want to use commercial off-the-shelf (COTS) systems or expand a product line.
- The Survivable Systems Initiative is a research area within the SEI that determines requirements for survivability.
- The Capability Maturity Model Integration (CMMI) project takes a holistic view of the requirements engineering challenge and thus touches on aspects of other SEI work.

In this article, we survey some of the ways in which the SEI is meeting the challenges of requirements engineering.

Shaping software architecture with quality requirements

Rick Kazman of the SEI's Architecture Tradeoff Analysis Initiative points out that there

are two kinds of requirements, functional and quality. Most customers and developers have focused on functional requirements--what the system does and how it transforms its input into its output. But while functional requirements are necessary, quality requirements are critical to the software architecture and significantly influence the shape of the architecture.

“You can satisfy a set of functional requirements with different software structures. The choice between these software structures is not constrained by the functional requirements,” Kazman says. “The choice is affected by how you meet the quality requirements. The way that I satisfy functional requirements with structure A versus structure B versus structure C may have a huge impact on how maintainable the system is, or how well it performs, or how well it scales, or how secure it is, or how reliable it is. Those qualities have nothing to do with what the system computes, but they determine how fit the system is for its purpose over time, how expensive it will be to build and maintain, and how error prone and secure it will be.”

Choices among different quality requirements shape the architecture, Kazman explains. “Each requirement suggests certain architectural structures and rules other ones out. I will choose one set of architectural structures over another because I know that it’s a good architecture for being able to predict and control end-to-end latency or throughput, or that it’s a good architecture for high availability.”

Architecture analysis provides another benefit to requirements-engineering efforts. The process, which can include structured walkthroughs of the architecture and construction of models, often clarifies a system’s requirements. “It causes us to ask questions of the requirements that often the stakeholders haven’t asked themselves,” Kazman says. For example, a requirement might call for end-to-end latency of three seconds. “It might turn out that it’s really hard to achieve that in a particular architecture. We might ask, ‘Do you really need three seconds?’ The stakeholders might say, ‘No, we just picked that number, it could be five.’ Or, they might say, ‘Yes, it’s three come hell or high water.’ Then we just have to change everything to meet three.”

Architecture analysis also uncovers missing, overlooked, and insufficiently well-understood requirements, as well as requirements that are too vague to be testable, such as: “The system shall be maintainable and robust.” Kazman says, “There’s no way you can test whether that requirement is met or not. The process forces you to say what you really mean by ‘maintainable and robust.’”

Using model-based verification to analyze requirements

Software requirements have been cited as the source of many if not most of the errors that are manifested throughout a development or upgrade effort, says Dave Gluch of the SEI's Dependable Systems Upgrade Initiative. Frequently these errors are not discovered until later, for example during design, code, or test, and they can remain latent until they cause a failure during operations.

Peer reviews of requirements can be effective in uncovering errors, but participants in reviews often spend much of their effort checking such properties as consistent terminology and syntax. “As a result of this attention to application-independent aspects, reviewers can fail to uncover many of the incorrect facts or logic errors associated with the application,” Gluch says. “In addition, many procedurally formalized review protocols do not provide technical direction for the reviewers, often resulting in subtle logic errors going undetected. Many of the errors in requirements arise out of complex interactions that cannot be easily unraveled through manual analysis.”

An emerging approach toward improving error identification throughout a software development or upgrade effort is to judiciously incorporate formal methods, in the form of models, into verification practices. The SEI is maturing and codifying this collection of techniques into a broad practice suite, termed “model-based verification,” for identifying certain types of errors in software designs.

At the center of model-based verification is a systematic practice of building and analyzing “essential” models of a system. Essential models are simplified formal

representations that capture the essence of a system, rather than provide an exhaustive, detailed description of it. “In analyzing essential models you don’t really execute the model.” Gluch says. “Rather, because it is a formal model, you explore its characteristics based on the mathematical properties of the model itself.”

The reduced complexity of essential models helps to provide the benefits of formal methods while minimizing the high cost normally associated with them. “We’re not saying do everything in formal methods. We’re saying use the formalism in a judicious and pragmatic way with essential models. Focus on what is essential to the system and model that.”

While model-based verification is applicable to the analysis of designs and code, the highest leverage and greatest benefits come from its use in analyzing requirements. “A good percentage of errors occur at the requirements stage, and if they’re not discovered until later on, the cost of fixing them is substantially higher,” Gluch says. “So the earlier you can find the error, the greater leverage you have. There can be an order of magnitude difference between finding an error in requirements versus finding it in design or in the code.”

Because it requires discipline and rigor to build a formal model, simply building the model uncovers errors in requirements. Then, once the formal model is built, it can be analyzed using automated model-checking tools, which can uncover especially difficult-to-identify errors that emerge in complex systems with multiple interacting and interdependent components. “These types of errors are almost impossible to detect during manual reviews,” Gluch says. “There have been cases where potentially catastrophic errors were uncovered in requirements specifications that had been extensively reviewed, simulated, tested, and implemented.”

An approach similar to model-based verification was used by the SEI for the POSIX 1003.21 working group during the early stages of developing a language-independent standard for real-time distributed systems communication. A mathematical specification language was used to create a model of the standard’s requirements. “The development and checking of the model--though without the benefit of tools--discovered gaps and inconsistencies in the requirements,” says the SEI’s Pat Place, who worked on the POSIX project. “The entire working group felt that the exercise was beneficial and was an efficient way to improve the quality of the requirements.”

A summary of model-based verification and its technical foundations can be found in the technical report, *Model-Based Verification: A Technology for Dependable Upgrade*.

The SEI has been conducting small-scale studies into the application of model-based verification techniques, focusing on the engineering and process issues associated with implementation. The results of one of the studies are summarized in the SEI technical report, *A Study of Practice Issues in Model-Based Verification Using the Symbolic Model Verifier (SMV)*.

These studies are providing important data on the time, expertise, and engineering decisions required, as well as insight into the effectiveness of the approach. The results of these studies and future pilot investigations on more complex real-world systems will form the basis for developing engineering and management guidelines for implementing and using model-based verification in complex software development and upgrade projects.

Specifying flexible requirements for COTS-based systems

In traditional software development, system requirements are defined in minute detail, and the system is then built to match those requirements. But if developers want to use commercial off-the-shelf (COTS) components, requirements often need to be much more flexible, and much less specific, says the SEI’s Pat Place. COTS components have usually been designed with the software marketplace, rather than a specific developer’s needs, in mind. “COTS products meet the requirements that are perceived to be the most likely to make a sale,” Place says. As such, if developers specify system requirements too narrowly, they may find that no COTS products exist that match those requirements.

One approach to achieving the necessary flexibility is to divide requirements into three groups: “must have,” “should have,” and “nice to have.” As the names suggest, systems that do not satisfy the “must-have” requirements are unsuitable, while the “should-have” and “nice-to-have” requirements become tradeoff points that help determine which COTS products should be used. “Partitioning the requirements into these groups involves a great deal of discipline,” Place says. The traditional view treats all requirements equally. Under the new appraoch, “each ‘must-have’ requirement needs to be examined and justified as to why it is non-negotiable.” Similarly, the Department of Defense’s “cost-as-independent-variable” principle states that if requirements make a system too costly, the requirements must be renegotiated.

Coupled with the notion of flexibility is the notion of abstraction: How much detail should be put into the requirements? “If we specify too much detail, we may be unable to find COTS products that match our requirements, thus eliminating the acquisition of a COTS-based system,” Place says. “However, too little detail means that we may be unable to distinguish between competing products that all claim to satisfy the requirements. Indeed, it may leave us unable to reject products that we know are unsuitable for use. In a competitive contracting situation it may be difficult to eliminate such proposals without fear of a protest.”

Product suitability also depends on quality factors, not just the satisfaction of functional requirements. For example, if the look and feel of the product’s user interface differs from the interface for other components of the system or embodies concepts unacceptable to the users--such as opaque windows when the users want transparent windows--then the interface is unsuitable. If the interface cannot be tailored to eliminate such problems, then the product itself is unsuitable.

One of the advantages of COTS-based acquisition is the existence of a marketplace of COTS products as well as a secondary marketplace of consultants. “It seems to be a general rule that, whenever there are a number of competing products, there will also be consultants willing to offer an opinion and help a purchaser select between the products,” Place says. “If we can identify honest consultants, then we might use those consultants to help set the requirements for our system. They can provide us with knowledge about different products’ capabilities, and they may also bring knowledge concerning previous attempts to use one product or another within a system. We have to be wary, though, since in addition to knowledge, consultants may bring bias toward one product or another.” This contrasts with traditional software development approaches, where consultants usually have been specialists in the system rather than the marketplace.

When an organization acquires a system with COTS components, it must consider the relationship between function and technology. Function defines what a product actually does, while technology refers to the underlying concepts of the product--whether it is client/server, distributed, or Web based, for example. “Many new technologies are available to us,” Place says. “If we specify a technology in the requirements, we may lock into a particular technological approach and eliminate many products in the marketplace. On the other hand, if we don't specify a technology, we may have to evaluate products embodying technologies that we would rather not use.” Another risk of not specifying technology is that developers might end up with a system whose technology is supported by only a limited number of vendors.

Further, just because a COTS product uses the right technology and provides the right function does not mean it is the best product for the intended system. “In at least one case, we’ve seen a COTS product chosen because it provided the right technological model--distributed client/server--with less attention paid to the details of the manner in which the product operated. When placed into the intended application, it was discovered that functionality was deficient and required extensive rework,” Place says. On the other hand, to achieve a desired functionality, developers might have to accept a particular technology, which could then affect the entire system’s architecture and design.

When acquiring a system with COTS components, requirements engineers must consider the overlap between the marketplace and the requirements for the system. “The requirements, marketplace, and design each influence the other. The result is that

a system developer must consider each of them simultaneously, accepting that the resultant system will be a compromise among these different concerns," Place says.

Developing asset and product requirements for product lines

Issues of flexibility also confront developers of software product lines and customers who want to apply product lines to new systems.

Developing software for product lines requires two separate requirements engineering processes, one for the assets that will be common across the product line and one for each individual product. Sholom Cohen, who works on the SEI's Product Line Practices Initiative, explains that the asset-development effort, usually called domain engineering, involves establishing boundary conditions for the product line. These include the functional and quality requirements, the classes of users, and the context for use.

If the asset requirements are thoroughly developed, the process for developing product requirements can be relatively simple. "Instead of needing to generate requirements from scratch, the new product could be 80 percent spelled out already," Cohen says. Some product line approaches use "generators." A user specifies the capabilities and requirements for a product and the software is automatically generated.

Determining requirements for product lines, especially lines that are expected to last for several years, can be extraordinarily difficult because it often involves forecasting the future. "You're not specifying requirements for products that are known entities," Cohen says. "You may be able to look at legacy systems, or at competitors' products. You may go to domain experts and technology experts." But, he points out, such efforts would not have helped a requirements engineer working in 1993 who wanted to plan a long-lasting set of products for the World Wide Web. "Today people are talking about total desktop computer systems for the Web that could not have been envisioned five or six years ago."

Cohen is currently working with a systems developer on a class of systems that have never been built, and which are expected to use future electronics that will go beyond anything that exists today. "The requirements problem becomes quite severe. It's a totally new concept that requires visiting a broad range of different types of stakeholders: users, developers, marketing people, and people who understand where the technology is going." He adds, "It's very hard to make this kind of prediction. But if you don't try to do it, you'll define the product too narrowly and come up with a rather static definition. If you define it too broadly, you end up with something that is too shallow to really cover anything in any depth. Then you have to go back and refine the requirements at a later stage."

For systems developers, using product lines offers great advantages in terms of lower costs, faster development times, and easy integration of new products. But developers have to be willing to work with the existing assets. "Our recommended approach is that the product requirements get developed in light of the existing assets, so the product line requirements, capabilities, and architectural qualities guide the new system development," Cohen says. "If some capabilities aren't there, maybe the developers can live without them, or they can be developed in a special way so that they are compatible with everything else in the product line."

In some cases, a new customer presents a set of requirements that raises questions about whether a product line has become obsolete. One company with which the SEI has worked, CelsiusTech, developed a product line for shipboard command and control that was installed on 10 systems for different navies around the world. When a new customer presented a requirements document, CelsiusTech determined that it would take three years and cost $10 million to build the system to specifications, but if the customer accepted only 85 percent of the capability the project could be done in six months for one-third the cost. "If a new customer says no, then the product line organization has to ask whether it's a request from out on the fringes, because of some unique weapons system or some geography they must deal with, or whether it represents a trend that's evolving. In that case, the organization would need to reconsider the pool of assets in its product line, and go through a new requirements cycle."

For more on the CelsiusTech case see the SEI technical report *A Case Study in Successful Product Line Management*.

Specifying requirements for survivability and mission support

The SEI and the software engineering community have recently begun to study the types of requirements that should be specified for software systems to survive adverse conditions and continue to support the organizational mission. Such requirements are especially important in large-scale, critical infrastructure systems, and in life- and mission-critical applications.

Survivability is defined as the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents. “People seldom, if ever, have a requirements specification for survivability,” says Nancy Mead of the SEI’s Survivable Systems Initiative. “They’ll talk about functional requirements and performance requirements. Sometimes they have security requirements, which is all about prevention. Survivability says, ‘What if I can’t prevent the attack or a failure?’” In the area of attacks and intrusions, Mead says software customers should consider a number of survivability questions:

- What is the impact on the organizational mission if the system is attacked and compromised?
- What parts of the system must continue to function during and immediately after an attack?
- How can the organization and the system recognize that there has been an attack, and how can it recover?

The SEI’s work on the Survivable Network Analysis (SNA) method has shown a real need to develop requirements for survivability. The prototype SNA method that is being used with clients enables software engineers to analyze an architecture to understand what functions must survive an attack, how to increase the survivability of those functions, and how to improve the capability of the system to recover from the attack. “We’ve got a long way to go in terms of understanding what it is that makes a good set of survivability requirements, and we’re doing research on that topic” Mead says. “Right now we can look at individual systems and say, after some analysis, ‘Here’s how you could strengthen the requirements in this area.’ But what we need is a general statement of what people should be doing.”

The SEI has held two IEEE-sponsored workshops on “information survivability,” which is a term often used interchangeably with “system survivability.” The most recent Information Survivability Workshop (ISW’98) was held in Orlando, Florida, this past October. Howard Lipson of the SEI served as general chair, and John Knight of the University of Virginia served as program chair. This invitation-only workshop focused on the domain-specific survivability requirements and characteristics of four different critical infrastructure and critical application areas: banking, electric power, transportation, and military information systems. The primary goal of the workshop was to foster cooperation and collaboration between domain experts and the survivability research community to improve the survivability of critical, real-world systems. ISW’98 brought together many of the leading researchers in the field of information survivability along with distinguished figures from critical infrastructure and application domains.

In its program of research and development, the Survivable Systems Initiative has also formed a Survivable Systems Working Group with Carnegie Mellon’s School of Computer Science to explore collaborative research efforts in survivability. And the SEI’s David Fisher and Howard Lipson are currently developing a survivable systems simulator.

The simulator promises to help organizations establish requirements and evaluate tradeoffs for survivable systems. Fisher and Lipson are using two assumptions to guide

their work:

1. No system is completely immune from an attack or disruption.
2. It is the system's mission that must survive, not any particular component of the system.

Lipson describes the survivability mission as "a very high-level statement of requirements" that is dependent on a particular context. For example, a financial system for trading on Wall Street might have a requirement that says it will never be down for more than five minutes. But what if a natural disaster interrupts power to New York City for 24 hours? Such a scenario encourages developers to think in terms of the context-dependent mission of the system, which in this case might be to maintain the integrity and confidentiality of data so that it can be quickly recovered after power is restored.

The ability to evaluate a system under "what-if" scenarios, such as the New York City power outage, is a central feature of the simulator. But rather than simply showing whether a system fails or succeeds, the simulator will provide information about how well the system survives. "It will show how robust the system is in the presence of the scenarios," Lipson says. This differs from traditional ideas about computer security, which are "binary"--an attack is either successful or it is not. With survivability, "there is no overall rating number, like '98 percent survivable.' It is all based on the context of survivability scenarios."

If the requirements of a system are not adequate for a certain level of survivability, they can be altered and run through the scenario again. Or, a different attack scenario can be run against the same set of requirements. Also, the requirements will not have to be precise and comprehensive. "The simulator will allow us to simulate applications at varying levels of abstraction," Fisher says. "Requirements tend to be abstract and not spelled out in every detail."

No simulator currently exists that can represent a system in a highly distributed, unbounded configuration, Fisher says. Traditional security approaches assume a bounded domain, in which an administrator has control of the entire system. Lipson adds, "The class of problems we're most interested in is associated with unbounded networks, such as the Internet. No one has complete control. Survivability is a new way of thinking that goes well beyond security."

The simulator is in the early development stages. Fisher and Lipson plan to demonstrate it internally at the SEI later this year.

Using technology to collaborate on requirements development

Another simulation is being tested by the SEI's Alan Christie, in cooperation with Mary Jo Staley, an SEI affiliate from Computer Sciences Corp. (CSC). The simulation is designed to determine how long it will take a team to develop requirements and how well those requirements will meet a particular quality standard.

Christie, who works on collaborative processes at the SEI, says he and his associate at CSC first developed a formal textual description of the requirements development process, based on CSC's principles for joint software application development. "It's a very intensive process," Christie says. "The team gets together for multiple-day sessions to tease out the issues associated with specific requirements for a particular application."

Christie's simulation assumes that the team will comprise three domain experts, a database expert, a graphical user interface expert, and a systems integration expert. The goal of the simulation is to predict--based on the amount of resources applied to the effort--how long the process will take and the quality of the requirements product that will emerge from the process. The simulation is exploring the impact of resource constraints at the organizational level with the effectiveness of member technical and communication skills at the detailed requirements development level. The output of this simulation provides insights into the length of the research and development process and the resulting requirements quality. The quality of the requirements in turn affects the subsequent review cycle, with potential upper management involvement if

quality is low.

Requirements engineering could also benefit from an investigation by Christie and the SEI's Ray Williams on collaborative team risk management. This project proposes to examine the difficulties encountered when teams of suppliers and customers must work together, over great geographic distances, to develop risk-management plans. The work is aimed at exploring the use of technologies to enhance communication, and the entire collaborative experience, for teams from different organizations and remote locations who must work together. The technologies, which go well beyond videoconferencing, include tools for collaborative brainstorming, such as computer-enhanced whiteboards that members can write on and view simultaneously at multiple locations, as well as projectors that display life-sized images of team members rather than showing them "on a monitor in the corner of the room," Christie says. Use of these technologies could foster more frequent and richer interactions, encourage a global perspective, and inhibit parochial and personal views from interfering with a collaborative effort.

Although the work would focus on team risk management, Christie says requirements development efforts "show very similar characteristics. You need the same sort of high-bandwidth communication channels to support both the computer displays that you can interact with in multiple locations and the large projected images." Also, he points out, requirements development often involves bringing together geographically dispersed supplier and customer teams. "We won't know what all the implications are until we gain more experience," he says. "But clearly, if you can allow people to interact frequently without them having to fly across the country, it has a real benefit."

Viewing requirements broadly in Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) project, a joint effort of the SEI, government, and industry, takes a holistic view of the requirements engineering challenge. As such, it touches on elements of the other requirements efforts reviewed in this article.

The objective of CMMI, which is expected to release models for public review and piloting in summer 1999, is to develop a product suite that provides industry and government with a set of integrated models to support organizational process improvement. It establishes common ground among systems engineering, software engineering, and the collection of maturity models that have evolved since the SEI's landmark release of the CMM for Software in 1993.

CMMI calls for organizations to be proactive and to think long term about requirements. "A lot of the work at the SEI, in areas such as architecture analysis, COTS, and product lines, has a direct benefit on the proactive character of the requirements process," says Mike Konrad, who is helping coordinate the CMMI project. "Your understanding of what the product should do matures with time." Konrad adds, "Other SEI initiatives play into the technical capability of the organization, and help it be more agile and more capable of understanding the implications of particular requirements." Those implications can include the product's competitive position, whether it faces regulatory issues, and its impact on the organization's long-term strategy. "That positioning and that proactive view are what we are focusing more attention on in CMM Integration," Konrad says.

CMMI also acknowledges the crucial role of communication among affected parties. "Many software and systems engineering organizations have learned that there must be a dialog for understanding what is required of the product, both now and in the future." says Mike Konrad.

But grounding that dialog in a language that is understandable to all parties is one of the thorniest challenges of requirements engineering, says the SEI's Mark Paulk, a principal developer of the CMM for Software. Often requirements are not explicitly stated, but are assumed, by either the end user or the requirements engineer, as part of the context. "As a specifier of requirements, I might not even know that we don't have a shared understanding until I actually get to the point of using the system that's been delivered," Paulk says. "It's a communication issue. You need domain experts working with the customer and the end user to help surface the things that users don't know

how to say or don't know that they need. It's very difficult."

Previous Capability Maturity Models touched on elements of requirements engineering, and requirements management is a key process area in the CMM for Software. But in the case of software embedded as part of a larger product that is to be produced, other requirements challenges were considered to be the domain of systems engineers--that is, another part of the developing organization--who had responsibility for system requirements, Konrad says. CMMI also recognizes that software and systems cannot be separated, and that software engineers must proactively work with systems engineers at the earliest stages of requirements capture and tradeoff analysis. "The two must work together because more and more products, which before had some kind of special-function mechanical or manufactured electronic component, now have an embedded piece of software that does that same function," Konrad says. "The software provides its own integration of many of the product functions that used to be handled separately by different subsystems. Those can now be integrated through software. Products are not just systems engineering and not just software engineering. The reality is that there have been lessons learned in both disciplines, and both software and systems engineers should benefit from having that shared understanding of the requirements process. CMMI will also benefit organizations for which the produced product is just software, enriching the information that has been given in the past with lessons learned from both software and systems engineering."

Ultimately, CMMI should help organizations develop "a deeper competence and understanding of what it takes to meet their customer's needs, now and in the future," Konrad says. "That knowledge may be a legacy of the requirements journey that all the other SEI initiatives play into."

About the author

Bill Thomas is the editor in chief of SEI Interactive. He is a senior writer/editor and a member of the SEI's technical staff on the Technical Communication team, where his duties include primary responsibility for the documentation of the SEI's Networked Systems Survivability Program.

His previous career includes seven years as director of publications for Carnegie Mellon University's graduate business school. He has also worked as an account manager for a public relations agency, where he wrote product literature and technical documentation for Eastman Kodak's Business Imaging Systems Division. Earlier in his career he spent six years as a business writer for newspapers and magazines.

He holds a bachelor of science degree in journalism from Ohio University and a master of design degree from Carnegie Mellon University.

Analyzing Quality Attributes

NEWS AT SEI

Author

Mario R. Barbacci

This library item is related to the following area(s) of work:

Software Architecture

This article was originally published in News at SEI on: March 1, 1999

In an earlier column, I compared the roles of a software architect and a traditional building architect. I suggested that this often-used analogy may not be entirely accurate, and that there may be a more precise analogy available to us in the emerging field of architectural engineering. In a subsequent release of *SEI Interactive*, Rick Kazman continued contrasting software architecture and "traditional" forms of architecture and engineering by describing the specific needs for representing software and system architectures. As noted by Kazman,

Engineering representations in any field serve to support analysis. Structural engineers, for example, might analyze the strength and bending properties of the materials that they use to construct a building, to determine if a roof will withstand an anticipated snow load, or if the walls will crumble in an earthquake.

A (software) system architecture must describe the system's components, their connections, their interactions, and the nature of the interactions between the system and its environment. Evaluating a system design before it is built is good engineering practice. A technique that allows the assessment of candidate architectures before the system is built has great value. As Winnie-the-Pooh would have it, "it would be a very good thing."

The architecture should include the factors or parameters of interest for each attribute model. Parameters that are common to more than one attribute model influence multiple attributes and can be used to trade off between attributes. For each of the attribute models, we must identify those parameters that have a major effect on the results for that model. A sensitive parameter is one that has a great impact on the

Related Links

News

SATURN Conference Announces Additional Keynote, Conference Scholarships

Distinguished Speakers, Strong Technical Program Set for SATURN 2014

See more related news »

Training

Big Data - Architectures and Technologies

Documenting Software Architectures - eLearning

See more related courses »

model. (Variations in the parameter correlate strongly with variations in the modeled or measured value.) Sensitive parameters found in only one set may not have been considered by the other model experts, or may not be relevant or sensitive to that model. Sensitive parameters that affect more than one attribute can be positively correlated [i.e., a change in one direction has positive effects on all attributes (win-win)] or negatively correlated [i.e., an improvement in one attribute may result in negative effects on another attribute (win-lose)].

A mature software engineering practice would allow a designer to predict these attributes through changes to the factors found in the architecture before the system is built. Unfortunately, in contrast to building architectures, we have yet to agree on what the appropriate software structures and views should be and how to represent them. One of the reasons for the lack of consensus on structures, views, and representations is that software quality attributes have matured (or are maturing) within separate communities, each with their own vernacular and points of view. For example, we studied the different schools/traditions concerning the properties of critical systems and the best methods to develop them [Barbacci 95]:

- performance--from the tradition of hard real-time systems and capacity planning
- dependability -- from the tradition of ultra-reliable, fault-tolerant systems
- security -- from the traditions of the government, banking, and academic communities
- safety -- from the tradition of hazard analysis and system safety engineering

Systems often fail to meet user needs (i.e., lack quality) when designers narrowly focus on meeting some requirements without considering the effect on other requirements or by taking them into account too late in the development process. For example, it might not be possible to meet dependability and performance requirements simultaneously:

Replicating communication and computation to achieve dependability might conflict with performance requirements (e.g., not enough time). Co-locating critical processes to achieve performance might conflict with dependability requirements (e.g., single point of failure). This is not a new problem, and software developers have been trying to deal with it for a long time, as illustrated by Boehm [Boehm 78]:

Finally, we concluded that calculating and understanding the value of a single overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict; added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness can conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations.

We should not look for a single, universal metric, but rather for quantification of individual attributes and for tradeoffs between different metrics. As we shall see later, identifying shared factors and methods gives us a good handle on the problem, provided we keep the relationships between attributes straight, which is not always easy.

Relationships between attributes

Each of the attributes examined has evolved within its own community. This has resulted in inconsistencies among the various points of view.

Dependability vis-a-vis safety

The dependability tradition tries to capture all system properties (e.g., security, safety) in terms of dependability concerns—i.e., defining failure as “not meeting

requirements" [Laprie 92]. It can be argued that this is too narrow because requirements could be wrong or incomplete and might well be the source of undesired consequences. A system could allow breaches in security or safety and still be called "dependable."

The safety-engineering approach explicitly considers the system context. This is important because software considered on its own might not reveal the potential for mishaps or accidents. For example, a particular software error may cause a mishap or accident only if there is a simultaneous human and/or hardware failure. Alternatively, it may require an environment failure to cause the software fault to manifest itself.

For example [Rushby 93], a mishap in an air-traffic control system is a mid-air collision. A mid-air collision depends on several factors:

- The planes are too close.
- The pilots are unaware that the planes are too close.
- Or the pilots are aware that the planes are too close, but fail to take effective evading action -- or are unable to take effective evading action
- The air-traffic control system cannot be responsible for the state of alertness or skill of the pilots; all it can do is attempt to ensure that the planes do not get too close together in the first place.

Thus, the hazard (i.e., the erroneous system state that leads to an accident) that must be controlled by the air-traffic control system is, say, "planes getting closer than two miles horizontally or 1,000 feet vertically of each other."

Precedence of approaches

Safe software is always secure and reliable--Neumann presents a hierarchy of reliability, safety, and security [Neumann 86]. Security depends on reliability (an attribute of dependability), and safety depends on security; hence, also reliability. A secure system might need to be reliable because a failure might compromise the system's security (e.g., assumptions about atomicity of actions might be violated when a component fails). The safety-critical components of a system need to be secure to prevent accidental or intentional alteration of code or data that were analyzed and shown to be safe. Finally, safety depends on reliability when the system requires the software to be operational to prevent mishaps.

Enhancing reliability is desirable, and perhaps necessary, but it is not sufficient to ensure safety. As Rushby notes [Rushby 93], the relationships are more complex than a strict hierarchy:

- Fault-tolerant techniques can detect security violations -- virus detected through N-version programming, intrusions detected automatically as latent errors, and denial detected as omission or crash failures.
- Fault containment can enhance safety by ensuring that the consequences of a fault do not spread and contaminate other components of a system.
- Security techniques can provide fault containment through memory protection, control of communications, and process walls.
- A security kernel can enforce safety using runtime lockin mechanisms for "secure" states and interlocks to enforce some order of activities. Kernelization and system interlocks are primarily mechanisms for avoiding certain kinds of failure and do very little to ensure normal service.

- A kernel can achieve influence over higher levels of the system only through the facilities it does *not* provide--if a kernel provides no mechanism for achieving certain behaviors, and if no other mechanisms are available, then no layers above the kernel can achieve those behaviors.

The kinds of behaviors that can be controlled in this way are primarily those concerning communication, or the lack thereof. Thus, kernelization can be used to ensure that certain processes are isolated from each other, or that only certain interprocess communication paths are available, or that certain sequencing constraints are satisfied.

Kernelization can be effective in avoiding certain faults of commission (doing what is not allowed) but not faults of omission (failing to do what is required)—that is, a security kernel cannot ensure that the processes correctly perform the tasks required of them.Applicability of approaches

The methods and mindset associated with each of the attributes that we examined [Barbacci 95] have evolved from separate schools of thought. Yet there appear to be common underpinnings that can serve as a basis for a more unified approach for designing critical systems. For example

- Safety and dependability are concerned with detecting error states (errors in dependability and hazards in safety) and preventing error states from causing undesirable behavior (failures in dependability and mishaps in safety).

- Security and performance are concerned with resource management (protection of resources in security and timely use of resources in performance).

The applicability of methods developed for one attribute to another attribute suggests that differences between attributes might be as much a matter of sociology as technology. Nevertheless, an attribute-specific mindset might be appropriate under certain circumstances. Examples include the following:

The dependability approach is more attractive in circumstances for which there is no safe alternative to normal service—a service *must* be provided (e.g., air-traffic control).

The safety approach is more attractive where there are specific undesired events--an accident *must* be prevented (e.g., nuclear power plant).

The security approach is more attractive when dealing with faults of commission rather than omission--service *must not* be denied, information must not be disclosed.

This is not to suggest that other attributes could be ignored. Regardless of what approach is chosen, we still need a coordinated methodology to look at all of these attributes together in the context of a specific design. For example, all the attributes that we examined [Barbacci 95] seem to share classes of factors. There are events (generated internally or coming from the environment) to which the system responds by changing its state. These state changes have future effects on the behavior of the system (causing internal events or responses to the environment).

The “environment” of a system is an enclosing “system,” and this definition applies recursively, up and down the hierarchy. For example varying arrival patterns (events) cause system overload (state) that leads to jitter (event); faults (events) cause errors (state) that lead to failure (events); hazards (events) cause safety errors that lead to mishaps (events); intrusions (events) cause security errors that lead to security breaches (events).

Architecture patterns are the building blocks of software architectures. Examples of patterns include pipes and filters, clients and servers, token rings, blackboards, etc. The architecture of a complex system is likely to include instances of more than one of these patterns, composed in arbitrary ways. Collections of architecture patterns should be evaluated in terms of quality factors and concerns, in anticipation of their use. That

is, it is conceivable that architecture patterns could be “pre-scored” to gain a sense of their relative suitability to meet quality requirements should they be used in a system.

In addition to evaluating individual patterns, it is necessary to evaluate compositions of patterns that might be used in an architecture. Identifying patterns that do not “compose” well (i.e., the result is difficult to analyze or the quality factors of the result are in conflict with each other) should steer a designer away from “difficult” architectures, toward architectures made of well-behaved compositions of patterns.

In the end, we will need both quantitative and qualitative techniques for evaluating patterns and architectures. Quantitative techniques include various modeling and analysis techniques, including formal methods. Scenarios are rough, qualitative evaluations of an architecture; scenarios are necessary but not sufficient to predict and control quality attributes and have to be supplemented with other evaluation techniques (e.g., queuing models, schedulability analysis). Architecture evaluations using scenarios will be the subject of a future column.

References

[Barbacci 95] Barbacci, M.R.; Klein, M.H.; Longstaff, T.A.; & Weinstock, C.B. *Quality Attributes* (CMU/SEI-95-TR-021, ADA307888). Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1995.

[Boehm 78] Boehm, B. et al. *Characteristics of Software Quality*. New York: American Elsevier, 1978.

[Laprie 92] Laprie, J.C. (ed.). *Dependable Computing and Fault-Tolerant Systems. Vol. 5, Dependability: Basic Concepts and Terminology* in English, French, German, Italian, and Japanese. New York: Springer-Verlag, 1992.

[Neumann 86] Neumann, P.G. “On Hierarchical Design of Computer Systems for Critical Applications.” *IEEE Transactions on Software Engineering* 12, 9 (September 1986): 905-920.

[Rushby 93] Rushby, J. *Critical System Properties: Survey and Taxonomy* (Technical Report CSL-93-01). Menlo Park, Ca.: Computer Science Laboratory, SRI International, 1993.

About the author

Mario Barbacci is a senior member of the technical staff at the SEI. He was one of the founders of the SEI, where he has served in several technical and managerial positions, including project leader (Distributed Systems), program director (Real-Time Distributed Systems, Product Attribute Engineering), and associate director (Technology Exploration Department). Before coming to the SEI, he was a member of the faculty in the School of Computer Science at Carnegie Mellon.

COTS Product Evaluation and System Design

NEWS AT SEI

Author

David J. Carney

This library item is related to the following area(s) of work:

System of Systems

This article was originally published in News at SEI on: March 1, 1999

In this column I will deal with one of the most interesting and often unexpected aspects of using commercial off-the-shelf (COTS) products in complex software systems. That is the relationship that exists between between product evaluation and system design. Numerous experiences have now shown that in systems that make extensive use of COTS products, there is a remarkable bond between evaluating the commercial components and designing the system. These two activities are not just closely related, but are in fact mutually dependent processes.

First Things First

I carefully used the expression "complex software system" in the opening paragraph, and that was an important point. We hear the phrase "COTS-based system" used quite often these days, and it can refer to anything from Microsoft Office to an avionics system that happens to run on a commercial operating system. The relationship I wish to consider is most pertinent for a certain subset of this wide expanse of system types, so it is first necessary to restrict the domain of discourse.

We note the existence of a spectrum of software systems that could be called "COTS based." At one extreme is the system that comes from a single vendor and that performs widely used and well-understood business functions. Examples abound in financial management, personnel management, and so forth. Such systems need tailoring, of course, but the notion is still one of "off the rack," in much the same the way that ready-made clothes are purchased. The customer chooses a vendor with products in the particular functional area; other discriminators might include cost, proximity, or past performance. The customer then describes more precisely what he or she is looking for, and the vendor responds with a solution that typically already

Related Links

Training

Migrating Legacy Systems to SOA Environments - eLearning

Service-Oriented Architecture: Best Practices for Successful Adoption

See more related courses »

exists. The vendor has previously solved the integration issues, has determined the system's broad architectural principles, and has resolved any questions of infrastructure. After some necessary tailoring for some specific needs (e.g., "we'll take it in an inch here"), the vendor supplies the system as a package to the customer.

At the other extreme is the system that an organization designs, constructs, and builds, a system that meets a specific and often unique need. The functionality of the system is the aggregate output of several--perhaps a great many--components, which have been integrated, typically with a good deal of "middleware" technology. The components themselves are obtained from a wide variety of sources, including commercial vendors and existing components, and some items are custom made for the occasion. Any issues such as incompatible interfaces between components are problems that the integrator must solve.

There are two major things that distinguish these extremes. One is that for the former class of system, the emphasis is on tailoring (or "customizing") while for the latter class of system the emphasis is on integration. The second distinguishing aspect is that for the former system, the conceptual "owner" of the system (regardless of who buys it) is the vendor who sells it as a package. It is the vendor to whom the customer turns if the system fails, who is generally expected to provide normal maintenance support, and so forth. In the latter kind of system, the conceptual owner is the organization that determined the functional characteristics, selected the constituent components, and integrated them into a coherent whole. Any maintenance is piecemeal: as new versions of each component appear, the organization chooses whether or not to upgrade that part of the system. These extreme ends of the "COTS-based system" spectrum are shown in the figure below. I refer to the former type as a "COTS-solution system" and the latter type as a "COTS-integrated system."

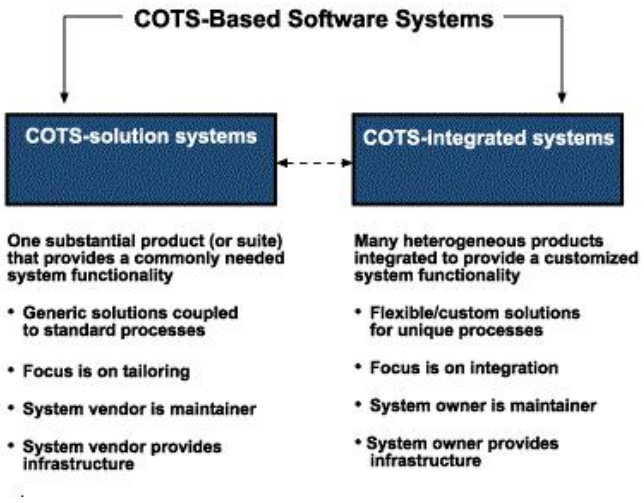

Figure 1: Extremes of the "COTS-based system" spectrum

For the remainder of this column, I shall concentrate on the right-hand side of this figure, on issues that arise in COTS-integrated systems, for it is in such systems that the mutual relationship between evaluation and design is most evident.

The Impact of a Commercial Approach on Design and Evaluation

For most people who have built software over the past few decades, the notions of design and evaluation are usually quite distinct, particularly in their chronology. Design is commonly done early, soon after requirements are defined; evaluation is commonly done fairly late in the process, whether in the sense of "test and evaluation," acceptance testing, or even, for some people, as a form of quality assurance. Design is done before the system exists, evaluation is done after it exists. You can't evaluate something until you've built it, and you can't build it until you design it. So far, so good. If, that is, you are building a system from scratch.

But when we choose to let more and more of a system come from pre-existing parts--whether because of a government mandate or because of hoped-for economies--the

way design and evaluation are performed changes in a subtle but very definite manner.

First, it becomes evident that the notion of "requirements" is now divided. On one hand, there is some collection of requirements that our final system must satisfy. On the other hand, each of the components that will be part of the system has some independent set of requirements that governed its creation. For commercial components, those requirements will rarely be explicit, and will almost certainly be based on marketplace imperatives. The detailed needs of our system are unknown to the COTS vendors, and even if they are known they are rarely of interest.

Second, these components--or many of them, at least--will already exist before our system is even specified. The components' interfaces, their architectural assumptions (e.g., choices between kernel-level threads or user-level threads and decisions about security factors), and their functional dependencies on other technologies will all characterize the products. They are not variable attributes that the system designer can change.

Third, the life cycle of individual components is in someone else's hands. Updates, revisions, changes to a component's internal architecture, even a decision to stop supporting a product are now all determined by the COTS vendor, and decisions are made as much for business reasons as by technical necessity.

A Merging of Design and Evaluation

These points force us to revise our notions of when and how we do evaluation and design. In traditional development, the principal constraint on a system may be its requirements. But a COTS-integrated system is constrained both by the system's requirements and by the capabilities and constraints of the available components. We cannot simply follow the traditional sequence of specifying requirements and designing a system, and then hope to perform the implementation phase simply by going out to buy COTS products. To do so will almost certainly be hopeless, because it assumes that somewhere in the COTS marketplace is a collection of commercial products that just happen to fit perfectly with our needs.

Instead, we now must do a significant amount of inspection of the available products *before* we solidify our design. We must do testing, benchmarking, prototyping--in short, we must include product evaluation as a part of the design process. (Even more difficult is trading off requirements against existing commercial products, but I shall deal with that in a later column.)

We find that this changes our existing notions of both design and evaluation. Our design activity has always included tradeoffs. But now the kinds of tradeoffs we make are much more various. Consider the "ilities," for example: As much as we consider the reliability of a component, we must also factor in the reliability of its vendor both to stay in business and to offer reasonable product support. As much as we assess the usability of a component, we must guess whether its vendor will still be in business three years hence.

The evaluation activity is equally changed. The traditional notion of evaluation was rooted in requirements: A "shall" statement existed for some throughput figure, and the software either did or did not perform adequately. Even for requirements that were not easily quantified, there was still the implicit assertion that they were satisfied in some specific manner. Now, however, COTS product evaluation has an added element of "what if?" We assess the requisite functional capabilities that we need, but we also look at what else the product might do. Serendipity is not necessarily a bad thing, and a product's unexpected features might lead us to reconsider the system's design. Sudden or unexpected marketplace developments, which seem to be more and more the rule, might also suggest system-design changes, an unhappy but pragmatic reality for the project manager.

And finally, these two activities are now carried out simultaneously. We start with certain design constraints, and we evaluate some products that might satisfy them. We then realize the implications that some products bring with them, and we modify the design to capitalize on a certain product's strengths. We make market forecasts about new products, and focus our design constraints accordingly. We do another form of

evaluation continuously, even apart from any particular system, by staying aware of current products and emerging technologies. What we *don't* do is leave the evaluation activity to the end of the development life cycle.

Summary

The effect of a COTS approach on software system development is not simply found in the system's cost. Savings may well result, but we should not forget Newton's rule about actions and opposite reactions. Realizing large savings from the COTS marketplace also means that we agree to be bound by the marketplace's realities. For those of us who specify and design systems, yet are also choosing to purchase the parts of those systems as COTS components, the marketplace will never provide exactly what we want, so we must make do with what we can find. System design and product evaluation jointly contribute to how we structure our systems. As I noted in the first paragraph, this fact is often unexpected. But it cannot be ignored.

My next column will continue with this topic and will provide some detailed examples of how product evaluation and system design can interact. Stay tuned.

About the Author

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

Requirements Engineering

NEWS AT SEI

This article was originally published in News at SEI on: March 1, 1999

Editor's Note: The following article is reprinted from the book *Software Requirements Engineering, Second Edition*, and is provided for readers who want to read a brief tutorial on requirements engineering. The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University.

Requirements engineering is presented and discussed as a part of systems engineering and software systems engineering. The need for good requirements engineering, and the consequences of a lack of it, are most apparent in systems that are all or mostly software. Requirements Engineering approaches are closely tied to the life cycle or process model used. A distinction is made between requirements engineering at the system level and at lower levels, such as software elements. The fundamentals of requirements engineering are defined and presented: elicitation; decomposition and abstraction; allocation, flowdown, and traceability; interfaces; validation and verification. Requirements development approaches, tools, and methods, and their capabilities and limitations, are briefly discussed.

Introduction

When the "Software Crisis"1 was discovered and named in the 1960s, much effort was directed at finding the causes of the now-familiar syndrome of problems. The investigations determined that requirements deficiencies are among the most important contributors to the problem: "In nearly every software project which fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure." [2] Development of the requirements specification "in many cases seems trivial, but it is probably the part of the process which leads to more failures than any other." [3]

It was determined that the benefits of good requirements include:

- Agreement among developers, customers, and users on the job to be done and the acceptance criteria for the delivered system
- A sound basis for resource estimation (cost, personnel quantity and skills, equipment, and time)
- Improved system usability, maintainability, and other quality attributes
- The achievement of goals with minimum resources (less rework, fewer omissions and misunderstandings)

It was also observed that the value of good requirements, and the criticality of doing them well, increased dramatically with the size and complexity of the system being developed. Additionally, software-intensive systems seemed to have more inherent complexity, that is, were more difficult to understand, than systems that did not contain a great deal of software; thus these systems were more sensitive to the quality of their requirements.

The products of a good requirements analysis include not only definition, but proper documentation, of the functions, performance, internal and external interfaces, and quality attributes of the system under development, as well as any valid constraints on the system design or the development process.

As the value of good requirements became clear, the focus of investigation shifted to the requirements themselves: how should they be developed? How can developers know when a set of requirements is good? What standards, tools, and methods can help; do they exist, or must they be developed? These investigations are by no means complete: not only are new tools and methods appearing almost daily, but overall approaches to requirements, and how they fit into the system life cycle, are evolving rapidly. As a result, requirements engineering has been well established as a part of systems engineering. Requirements engineers perform requirements analysis and definition on specific projects as well as investigate in the abstract how requirements should be developed.

Requirements engineering and the development life cycle

Many models exist for the system and/or software life cycle, the series of steps that a system goes through from first realization of need through construction, operation, and retirement.4 (Boehm5 provides a good overview of many existing models, and presents as well a risk-driven approach that includes many other models as subsets; Davis et al.6 describe conditions under which various models might be used.) Almost all models include one or more phases with a name like “requirements analysis” or “user needs development.” Many models require generation of a document called, or serving the function of, a requirements specification. Even those that do not call for such a document, for example Jackson System Development, have a product such as a diagram or diagrams that incorporate or express the user’s needs and the development objectives.7

A few of the better-known life cycle models are briefly discussed in the following sections, and the way requirements engineering fits into them are presented.

Baseline management

Among the most extensively used models are baseline management and the waterfall, on which baseline management is based.8 (baseline management differs from the waterfall in that it specifically requires each life cycle phase to generate defined products, which must pass a review and be placed under configuration control before the next phase begins.) In these models, as shown in Figure 1, determination of requirements should be complete, or nearly so, before any implementation begins. baseline management provides a high degree of management visibility and control, has been found suitable for developments of very large size in which less complex methods often fail, and is required under many military standards and commercial contracts. This model, however, has been somewhat discredited, because when large complex systems are developed in practice it is usually impossible to develop an accurate set of requirements that will remain stable throughout the months or years of development that follow completion of the requirements. This essential and almost unavoidable

difficulty of the waterfall and baseline management models had been noted for many years9, 10 but was brought to the attention of the U.S. defense software community by a Defense Science Board report authored by F. Brooks.11 Brooks pointed out that the user often did not know what the requirements actually were, and even if they could be determined at some point in time they were almost certain to change. To resolve this problem Brooks recommended an evolutionary model, as is discussed below. The approach advocated by Brooks provides the following advantages:

- The user is given some form of operational system to review (a prototype or early evolution), which provides better definition of the true needs than can be achieved by reading a draft specification. This approach avoids what Brooks identified as the unacceptable risk of building a system to the a priori requirements.
- Delivery of some operational capabilities early in the development process—as opposed to the delivery of everything after many months or years—permits incorporation of new requirements and of capabilities that did not exist or were not feasible at the start of development.

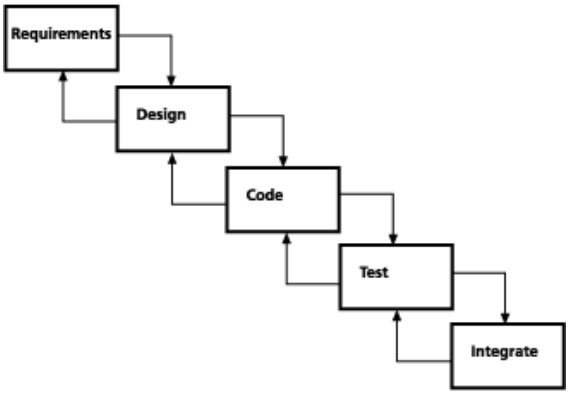

Figure 1: The baseline management and waterfall models

Prototyping

The prototyping life cycle (Figure 2) is one approach to the use of an operational system to help determine requirements.12 In this model, some system capability is built with minimum formality and control to be run for or by the user, so that requirements can be determined accurately. Several successive prototypes will usually be built. The amount of requirements analysis that precedes prototyping depends on the specifics of the problem. It is normally recommended that the prototype should be used only to help generate a valid set of requirements; after the requirements are available, they should be documented, and development should proceed as in the baseline management model. If this recommendation is followed, the prototyping portion of the life cycle may be considered as a tool or method supporting requirements analysis within the baseline management model.

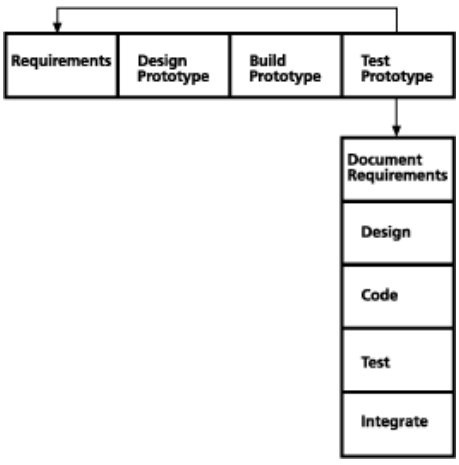

Figure 2: The Prototyping life cycle model

Many tools and methods are available to help support prototyping. The term *rapid prototyping* is associated with some of them, to distinguish them from other forms, such as development of high-risk hardware and software components, which is a slow, expensive process. Much of rapid prototyping is concentrated in two application areas: user interfaces and heavily transaction-oriented functions such as database operations. In these areas, a distinction can be made between prototyping tools and approaches that provide only "mockups" (simulate the system's response to user actions) and those that actually perform the operations requested by the user. In the latter category are the so-called fourth generation languages (4GLs),13, 14 which provide methods of generating code for user operations included within the 4GL's capability. Such tools provide the option of retaining the prototype as (part of) the final system; considerations of execution time and efficiency of memory usage are weighed against the time and cost of building a system using the baseline management model and requirements determined from the rapid prototyping effort.

Incremental development

The incremental development life cycle model calls for a unitary requirements analysis and specification effort, with requirements and capabilities allocated to a series of increments that are distinct but may overlap each other (Figure 3). In its original conception the requirements are assumed to be stable, as in the baseline management model, but in practice the requirements for later increments may be changed through technology advancement or experience with the early deliveries; hence this model may in effect be not very different from the evolutionary development model described next.

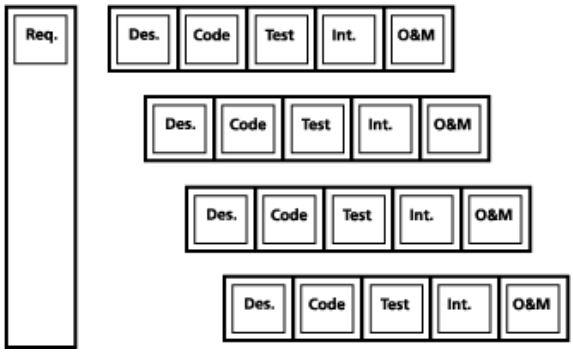

Figure 3: The incremental development model

Evolutionary development

The evolutionary development life cycle calls for a series of development efforts, each of which leads to a delivered product to be used in the operational environment over an extended period of time (Figure 4). In contrast with the prototyping model, in which the purpose of each early product is only to help determine requirements, each delivery or evolution provides some needed operational capability. However, there is feedback from users of the operational systems that may affect requirements for later deliveries.

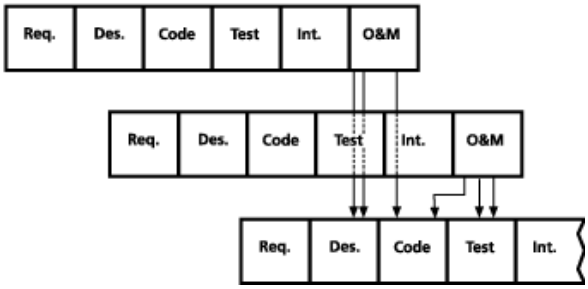

Figure 4: The evolutionary development model

Each delivery in this model represents a full development cycle, including requirements analysis. The deliveries may overlap, as shown in Figure 4, or one

delivery may be completed before the next is begun. The product of each requirements analysis phase is an addition or improvement to the product(s) of the requirements analysis phase of the previous delivery. Similarly, the implementation portions of each delivery may add to, or upgrade, products of earlier deliveries. With this understanding, each delivery may be looked at as a small example of a baseline management life cycle, with a development process and time span small enough to minimize the problems discussed above.

The spiral model

Boehm [5] describes the spiral model, an innovation that permits combinations of the conventional (baseline management), prototyping, and incremental models to be used for various portions of a development. It shifts the management emphasis from developmental products to risk, and explicitly calls for evaluations as to whether a project should be terminated. Figure 5 summarizes the spiral model.

The radial coordinate in Figure 5 represents total costs incurred to date. Each loop of the spiral, from the $(-x)$ axis clockwise through 360 degrees, represents one phase of the development. A phase may be specification oriented, prototyping oriented, an evolutionary development step, or one of a number of other variants; the decision on which form to use (or whether to discontinue the project) is made at each crossing of the $(-x)$ axis by evaluating objectives, constraints, alternatives, and status (particularly risk).

The spiral model thus makes explicit the idea that the form of a development cannot be precisely determined in advance of the development: the re-evaluation at the completion of each spiral allows for changes in user perceptions, results of prototypes or early versions, technology advances, risk determinations, and financial or other factors to affect the development from that point on.

Boehm has referred to the spiral model as a "process model generator": given a set of conditions, the spiral produces a more detailed development model.15 For example, in the situation where requirements can be determined in advance and risk is low, the spiral will result in a baseline management approach. If requirements are less certain, other models such as the incremental or prototyping can be derived from the spiral. And, as mentioned above, re-evaluation after each spiral allows for changes in what had been (tentatively) concluded earlier in the development.

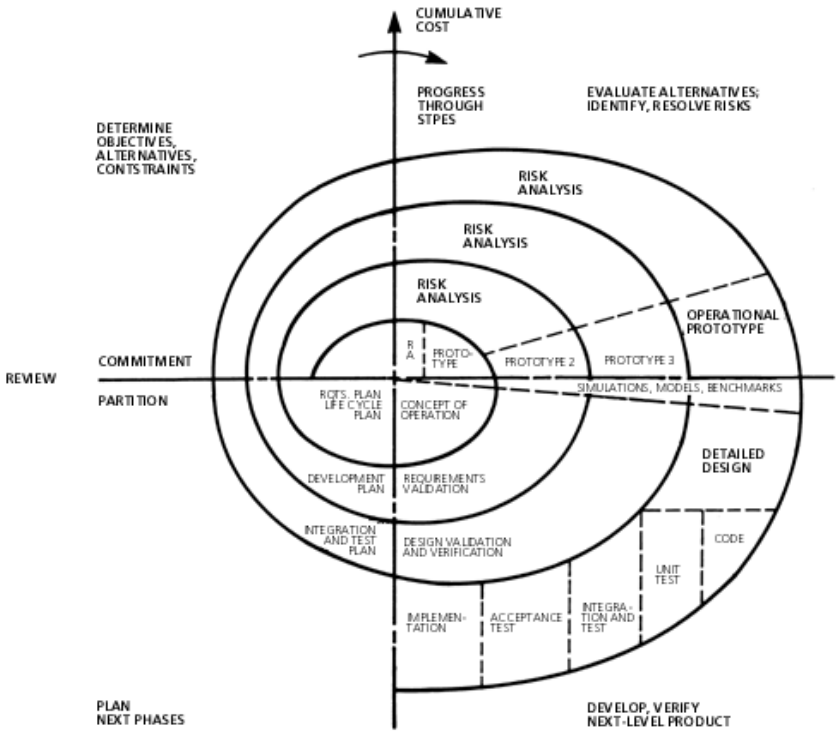

Figure 5: The spiral model

System and software requirements

System and software requirements are often treated together because the tools and methods used to derive them, and the techniques of documenting them, are very similar. (It may be remarked that most of the tools and methods originated with software developers and were then found to be appropriate for system use as well.) However, some important differences between system and software requirements should be pointed out.

System requirements describe the behavior of the system as seen from the outside, for example, by the user. Although requirements documents and specifications cannot easily be read by users,16 the system requirements serve at least as a partial communications vehicle to the more technically inclined users, or to a purchasing organization, as well as to analysts and designers who are concerned with the development of system elements or components.

Requirements for elements below the system level, whether they are for elements that are all hardware, all software, or composite (both hardware and software), are normally of minimal interest to users. These requirements serve to communicate with the developers, who need to know what is expected of the elements for which they are responsible, and who also need information about those elements with which they must interface.

This distinction is important for several reasons. First, like any communications vehicle, a requirements document should be written with its intended audience in mind. The degree to which nontechnical users must read and understand a requirements document is a factor in how it is written. Second, and perhaps most important, the skills and experience of the requirements developers must be considered. All too frequently, systems engineers with limited software knowledge are responsible for software-intensive systems. They not only write system-level requirements, which demands knowledge of what functions and performance a software-intensive system can be expected to meet; they also allocate functions and requirements to hardware and software. Thus the software developers are asked to develop designs to meet requirements that may be highly unrealistic or unfeasible. Software developers seem to be reluctant to get involved in systems engineering; one of the results is that the development of software elements often starts with poor requirements.

The U.S. Air Force Aeronautical Systems Center has recognized the importance of systems engineering to a software development by including in its software development capability evaluation (SDCE)17 material about systems engineering capability and its interface with software engineering. The SDCE is an instrument used to help acquisition organizations determine whether bidders who have submitted proposals for a software contract are likely to be able to perform acceptably.

Section IV carries further the distinction between system and software requirements as the approach to generating requirements for all system elements is outlined.

Fundamentals of requirements engineering

This section presents the overall framework within which requirements engineering takes place. Information about tools and methods is not presented here; at this point concern is focused on the sequence of events.

Several taxonomies have been proposed for requirements engineering. Prof. Alan Davis has proposed the following:[18]

- Elicitation
- Solution determination
- Specification
- Maintenance

Another is that requirements engineering consists of elicitation, analysis, specification,

validation/verification, and management. The comparison with Davis's is straightforward—validation/verification is included in maintenance, and management is implicit in all four of Davis's activities.

A system of any but the smallest size will be decomposed into a hierarchy of elements. Starting with the lowest levels, the elements are integrated into larger-size elements at higher levels, back up to the full system (Figure 6). Several approaches are available for development of the hierarchy, but all produce definitions of elements at all levels of the hierarchy. In the functional approach (implied by the element names used in Figure 6), the elements represent the parts of the system that will meet particular system requirements or carry out particular system capabilities. In a physical decomposition, the elements will represent physical components of the system. In a data-driven approach, the components will contain different parts of the key data needed by the system, and most likely also the operations carried out on that data. In an object-oriented approach, the components will consist of *objects* that include not only physical components of the system but also the data and functions (operations) needed by those physical components.

After the lowest-level elements (*units* in Figure 6) are defined, they are separately developed and then integrated to form the next larger elements (*programs* in Figure 6). These elements are then integrated into larger-size elements at the next level, and so on until the entire system has been developed, integrated, and tested. Thus the life cycle models presented earlier can be seen to be oversimplified in that they do not account for the development and integration of the various elements that compose the system. This aspect can be considered to be outside the responsibility of the requirements engineer, but, as discussed below, it affects cost, feasibility, and other factors that bring the realities of implementation to the development of requirements, in contrast to the strict separation of "what" and "how" often postulated for system development.

The requirements engineer needs to be concerned with all the requirements work that takes place. As described above, this includes requirements for the entire system, for composite (hardware and software) elements at lower levels, and for all-hardware and all-software elements. It should be noted at this point that references to *the* "requirements specification" are meaningless for any but the smallest systems; there will be several or many requirements specifications.

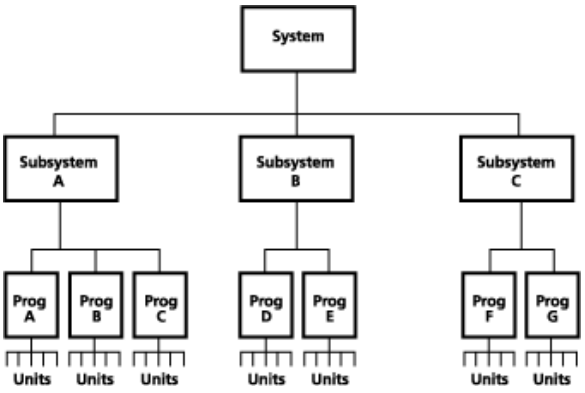

Figure 6: Example system hierarchy

System requirements

Next is described the mechanics of filling out the system hierarchy with requirements.[19, 20] It should be pointed out that, although the hierarchy used as an example is functional, the process, and the traceability aspects that accompany it, are valid whether the decomposition is functional, physical, data driven, or object oriented.

Early in the development process, the system-level requirements are generated. A primary tool used in generating the system requirements is the Concept of Operations or ConOps document,[16] a document that is narrative in form and describes the environment and operation of the system to be built. An important part of the development of both the ConOps and the system requirements is the process of

requirements elicitation, which is defined as working with the eventual users of the system under development to determine their needs.[21] Elicitation involves an understanding of psychological and sociological methods as well as system development and the application domain of the system to be built.

While the system requirements are being developed, requirements engineers and others begin to consider what elements should be defined in the hierarchy. By the time the system requirements are complete in draft form, a tentative definition of at least one and possibly two levels should be available. This definition will include names and general functions of the elements. Definition of the system hierarchy is often referred to as *partitioning*.

Allocation

The next step is usually called *allocation*. Each system-level requirement is allocated to one or more elements at the next level; that is, it is determined which elements will participate in meeting the requirement. In performing the allocation, it will become apparent that (1) the system requirements need to be changed (additions, deletions, and corrections), and (2) the definitions of the elements are not correct. The allocation process therefore is iterative, leading eventually to a complete allocation of the system requirements, as shown in Figure 7. The table in Figure 7 shows which element or elements will meet each system requirement; all requirements must be allocated to at least one element at the next level. In this example, we have called the level below system the Subsystem level; in practice, the names are arbitrary, depending on the number of levels in the entire hierarchy and the conventions in use by the systems engineering organization. Figure 7 shows that the system-level requirement denoted as SYS001 is allocated to subsystems A and B, SYS002 is allocated to A and C, and so forth.

SYSTEM REQUIREMENTS	SUBSYSTEM A	SUBSYSTEM B	SUBSYSTEM C
SYS 001	X	X	
SYS 002	X		X
SYS 003		X	
SYS 004	X	X	X
SYS 005			X
SYS 006	X	X	
SYS 007			

Figure 7: Example of allocation of system requirements

Flowdown

The next step is referred to as *flowdown*. (The reader should be aware that this nomenclature is not universal.) Flowdown consists of writing requirements for the lower-level elements in response to the allocation. When a system requirement is allocated to a subsystem, the subsystem must have at least one requirement that responds to that allocation. Usually more than one requirement will be written. The lower-level requirement(s) may closely resemble the higher-level one, or may be very different if the system engineers recognize a capability that the lower-level element must have in order to meet the higher-level requirement. In the latter case, the lower-level requirements are often referred to as *derived*.

The level of detail increases as we move down in the hierarchy. That is, system-level requirements are general in nature, while requirements at low levels in the hierarchy are very specific. A key part of the systems engineering approach to system development is *decomposition* and *abstraction:* the system is partitioned (decomposed) into finer and finer elements, while the requirements start at a highly abstract (general) level and become more specific for the lower-level elements. Large software-intensive systems are among the most logically complex of human artifacts,

and decomposition and abstraction are essential to the successful management of this complexity.

When flowdown is done, errors may be found in the allocation, the hierarchy definition, and the system requirements; thus the flowdown process is also iterative and may cause parts of previous processes to be repeated. Figure 8 shows the results of the first level of the flowdown process: a complete set of requirements for each of the subsystems. System-level requirement SYS001 was allocated to subsystems A and B; subsystem requirements (in this example, SSA001, SSA002, and SSB001) are written in response to the allocation. Similarly SYS002 was allocated to A and C, and subsystem requirements SSA003, SSA004, SSA005, SSC001, and SSC002 are the flowdown of SYS002 to subsystem levels. The result is a complete set of requirements for each of the subsystems. After completion of this level of flowdown, allocation of the subsystem requirements is carried out to the next level, followed by flowdown to that level. Again the processes are iterative and changes may be needed in the higher-level definition, allocation, or flowdown.

SYSTEM REQUIREMENT	SYSTEM A REQUIREMENT	SYSTEM B REQUIREMENT	SYSTEM C REQUIREMENT
SYS 001	SSA 001 SSA 002	SSB 001	—
SYS 002	SSA 003 SSA 004 SSA 005	—	—
SYS 003	—	SSB 002 SSB 003	—
SYS 004	SSA 006 SSA 007	SSB 004 SSB 005 SSB 006	SSC 003
SYS 005	—	—	SSC 004 SSC 005
SYS 006	—	SSB 007 SSB 008	—
SYS 007	SSA 008 SSA 009	SSB 009	—

Figure 8: Example of flowdown of system requirements

The process of partitioning, allocation, and flowdown is then repeated to as low a level as needed for this particular system; for software elements this is often to the module level. Figure 9 emphasizes the iterative nature of this process at each level in the many levels of partitioning, allocation, and flowdown.

good-citations-formatted.background_mar99_files/image007.gif"

Figure 9: Iteration in partitioning, allocation, and flowdown

Traceability

The number of requirements proliferates rapidly during the allocation and flowdown process. If we assume that there are four levels in the hierarchy, that each element partitions to four at the next level, that each requirement is allocated to two of the four, and that each flowdown results in three requirements per allocated element (all reasonable assumptions), there will be more than 250 requirements in the hierarchy for each system-level requirement. Keeping track of all these requirements is essential, to make sure that all requirements are properly flowed down to all levels, with no requirements lost and no “extras” thrown in. Reading and understanding the requirements to answer these questions is difficult enough; without a way to keep track of the flowdown path in a hierarchy of thousands of requirements, it becomes impossible. Traceability is the concept that implements the necessary bookkeeping.[19,

Establishment of traceability as allocation and flowdown are done helps ensure the validity of the process. Then, if changes are needed (such as a system-level requirement due to user input, or a lower-level requirement due to a problem with allocation, flowdown, or feasibility), traceability enables the engineer to locate the related requirements, at higher and lower levels, that must be reviewed to see if they need to be changed.

Figure 10 shows the traceability path corresponding to the allocation and flowdown in Figures 7 and 8. System requirement SYS001 traces downward to SSA001, which in turn traces downward to PGA001, PGA002, and PGB001. SYS001 also traces to SSA002, which further traces to PGA003, PGC001, and PGC002. Upward traceability also exists, for example, PGB001 to SSA001 to SYS001. A similar hierarchy exists through SYS001's allocation and flowdown to subsystem B. Other formats such as trees and indented tables can be used to illustrate traceability, as in Dorfman and Flynn.[19]

Note that, while allocation and flowdown are technical tasks, traceability is not strictly an engineering function: it is a part of requirements management and is really only "bookkeeping." A case can be made that more of the requirements problems observed in system development are due to failures in requirements management than to technical functions. In the Software Engineering Institute's Capability Maturity ModelÒ for Software,22 these nontechnical aspects of requirements management are important enough that they are one of the six key process areas that a software development organization must satisfy to move beyond the first ad hoc or chaotic level of maturity.

SYSTEM REQUIREMENT	SUBSYSTEM A REQUIREMENT	PROGRAM a REQUIREMENT	PROGRAM B REQUIREMENT	PROGRAM C REQUIREMENT
SYS 001	SSA 001	PGA 001 PGA 002	PGB 001	—
	SSA 002	PGA 003	—	PGC 001 PGC 002
SYS 002	SSA 003	—	PGB 002 PGB 003	—
	SSA 004	PGA 004	PCB 004	PGC 003
	SSA 005	PGA 005	PCB 005 PCB 006	PGC 004 PGC 005

Figure 9: The requirements traceability path

Interfaces

An additional step is interface definition. Before development of system requirements can begin, the system's external interfaces (the interfaces between the system and the outside world) must be known. As each level of partitioning, allocation, and flowdown takes place, the interfaces of each element to the rest of the system must be specified. This definition has two parts. First, interfaces defined at higher levels are made more specific; that is, the external interfaces to the entire system are identified as to which subsystem(s) actually perform the interface. Second, internal interfaces at that level are defined, that is, the subsystem-to-subsystem interfaces needed to enable each subsystem to meet the requirements allocated to it.

Figure 11 illustrates this concept. In the top diagram, A represents an external interface of the system, for example, an output produced by the system. When subsystems 1, 2, 3, and 4 are defined, as shown in the lower diagram, A is identified to subsystem 1, that is, the output originates in subsystem A, and internal interfaces, such as B between 3 and 4, are found to be necessary. This process continues throughout development of the hierarchy.

It is also possible that errors in partitioning, allocation, and flowdown will be discovered when interface definition is taking place, leading to iteration in those earlier steps.

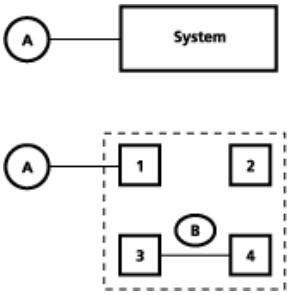

Figure 11: Increasing detail of interface definition

The refined life cycle model

Next, let's examine the implications of the above processes for the life cycle models discussed earlier. It should now be apparent that the single "requirements analysis" phase, even if extended to "system requirements analysis" and "software requirements analysis," is inadequate. The life cycle model should account for the multiplicity of levels in the hierarchy, and furthermore should recognize that the various subsystems and other elements at any level do not need to be synchronized. That is, if we are willing to accept the risk that flowdown to the last subsystem will surface some errors in partitioning or allocation, we can phase the subsystems, and of course the lower-level elements as well.

Figure 12 is the more realistic, and more complicated, life cycle chart that results from the above considerations. Although it builds on a baseline management approach, the nesting and phasing shown will apply to any of the models discussed earlier, or to combinations. The key features of Figure 12 are as follows:

1. For all but the lowest level of the hierarchy, the implementation phase of its life cycle becomes the entire development cycle for the next lower elements. Thus the subsystems have their development cycle shown as a phase of the system life cycle.
2. Elements at any level may be phased. Thus the subsystems are shown as starting and finishing their development cycles at different times. This approach has at least two advantages:
3. Staff can be employed more efficiently, since the schedules for each element can be phased to avoid excessive demand for any technical specialty at any time.
4. Integration can be carried out in a logical fashion, by adding one element at a time, rather than by the "big bang" approach of integrating all or many elements at the same time.

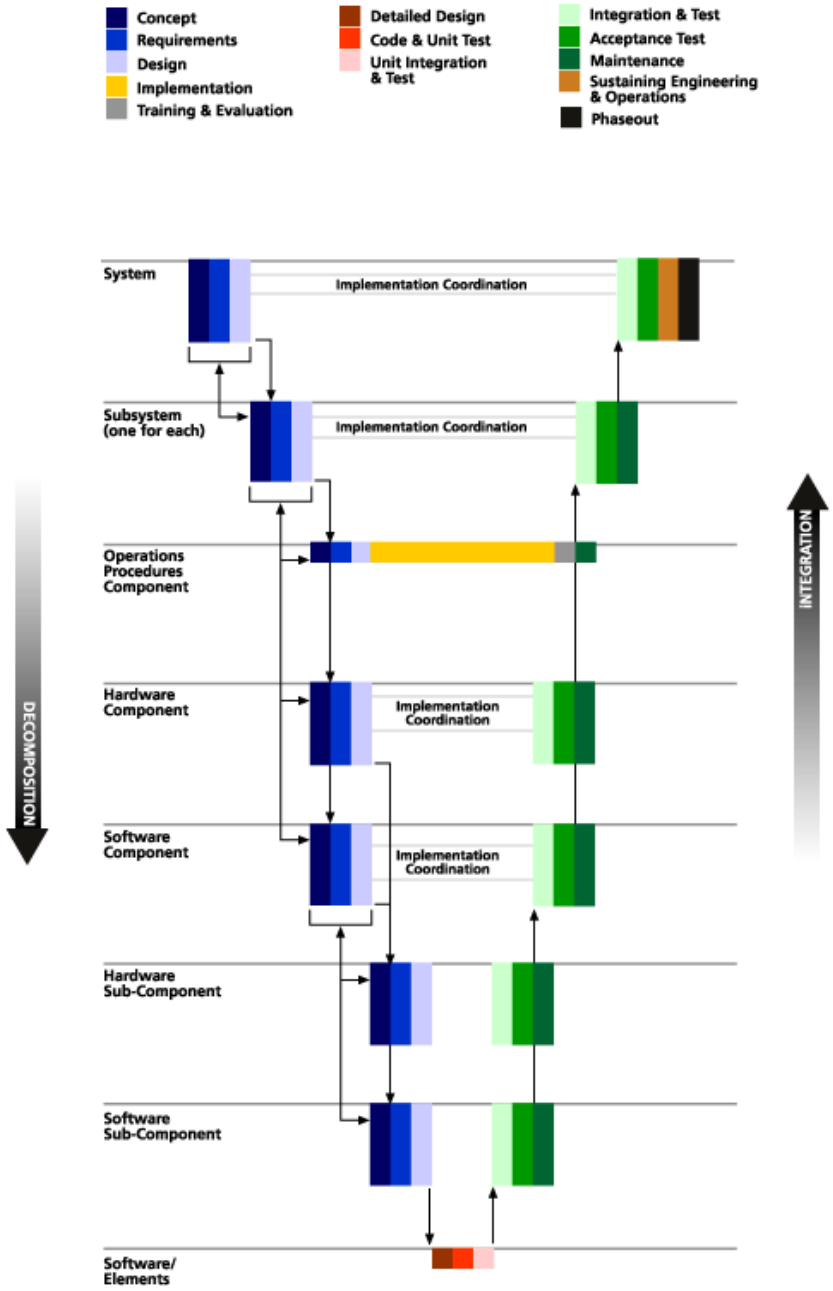

Figure 12: Multilevel life cycle chart

Validation and verification

A final point in the framework of fundamentals relates to another requirements management task, the review of the requirements.23 Validation and verification of the partitioning, allocation, flowdown, and interfaces are equally as important as their generation. It has been shown repeatedly that requirements errors not found until later in the development cycle are many times more expensive to fix than if they were found before the requirements phase was completed.24 The baseline management model requires that all the requirements at all levels be fully and properly reviewed before design and implementation begin. The life cycle models wherein the requirements are not all determined and “frozen” before design begins specify review of requirements as they are considered ready to be the basis for some design and further development.

The attributes of a good requirements specification are addressed by Boehm.[23] Among the most important attributes are the following:

- Clear/unambiguous
- Complete

Correct

- Understandable
- Consistent (internally and externally)
- Concise
- Feasible

It should be apparent that the evaluation of a requirements specification with respect to these attributes may be highly subjective and qualitative. Davis et al.25 demonstrate the extreme difficulty of attempting to quantify the degree to which a specification exhibits these qualities. Nevertheless, these attributes are so important that the verification and validation of requirements against these criteria must be carried out, to the degree possible and as quantitatively as possible. Boehm[23] and Davis et al[25] address approaches to validation and verification of requirements.

Requirements engineering and architectural design

In the previous section, an overview is given of the process of partitioning, allocation, and flowdown. The result of this process (a definition of the hierarchy down to some level, generation of the requirements for all elements, and determination of the interfaces between them) is known as the *architectural design* or *top-level design* of the system. Although it is called a design, requirements engineering is involved throughout the process. What, then, is the distinction between requirements analysis and design in this process?

Requirements analysis is often defined as the “what” of a problem: implementation free; containing objectives, not methods. Design, then, is the “how”: the implementation that will meet the requirements. The two are supposed to be kept distinct, although of course the feasibility of meeting a requirement always needs to be considered. However, if you look closely at the process of generating the architectural design, you will see that both requirements analysis and design are involved.26

The generation of system-level requirements is, to the extent possible, a pure “what,” addressing the desired characteristics of the complete system. The next steps, determining the next level of the hierarchy and allocating system requirements to the elements, are in fact a “how”: they do not address objectives beyond the system requirements, but they define a subsystem structure that enables the requirements to be met. Flowdown is again a “what,” determining what each element should do (functions, performance, and so on).

Development of the architectural design is, then, a process in which the steps of requirements analysis and design alternate, with more detail being brought out at each cycle. The output of requirements analysis is input to the next stage of design, and the output of design is input to the next stage of requirements analysis.27 If different people perform the two functions, one person’s requirement is the next person’s design, and one person’s design is the next person’s requirements.28

Pure requirements analysis, like pure design, can only go so far. Both disciplines are needed to achieve the desired result, a system that meets its user’s needs. Note that the character of requirements analysis, like that of design, changes as we move down in the hierarchy: requirements analysis for a low-level element is much more detailed, and involves knowledge of previous design decisions. The tools and methods used for “analysis” do in fact support all aspects of architectural design development: partitioning, allocation, and flowdown. They therefore are useful in both the requirements analysis and design stages of the process.

Requirements engineering practices

The principles of requirements engineering described above are valid and important, but for practical application additional specifics are needed. These specifics are provided by methods and tools. A *method*, sometimes referred to as a *methodology*, describes a general approach; a *tool*, usually but not always automated, provides a detailed, step-by-step approach to carrying out a method.

Methods

Requirements analysis methods may be roughly divided into four categories, as shown in Figure 13. The categorizations should not be regarded as absolute: most methods have some of the characteristics of all the categories, but usually one viewpoint is primary.

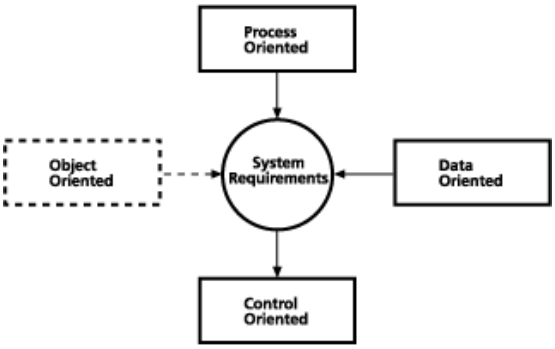

Figure 13: Categories of requirements analysis methods

Process-oriented methods take the primary viewpoint of the way the system transforms inputs into outputs, with less emphasis on the data itself and control aspects. Classical structured analysis (SA)29 fits into this category, as do structured analysis and design technique (SADT),30 and formal methods such as VDM31 and Z.32

Data-oriented methods emphasize the system state as a data structure. While SA and SADT have secondary aspects of the data viewpoint, entity-relationship modeling33 and JSD [7] are primarily data oriented.

Control-oriented methods emphasize synchronization, deadlock, exclusion, concurrence, and process activation and deactivation. SADT and the real-time extensions to SA34, [35] are secondarily control oriented. Flowcharting is primarily process oriented.

Finally, object-oriented methods base requirements analysis on classes of objects of the system and their interactions with each other. Bailin36 surveys the fundamentals of object-oriented analysis and describes the variations among the several different methods in use.

Tools

The number of tools that support requirements engineering is growing rapidly, and even the most cursory survey is beyond the scope of this paper. Nevertheless, some discussion of the characteristics of requirements engineering tools, and trends in these characteristics, is in order.

Davis [18] has classified requirements tools as follows:

- Graphical editing
- Traceability
- Behavior modeling
- Databases and word processing—not designed for requirements engineering, but used in requirements applications
- Hybrid (combinations of the above)

Early requirements tools, such as SREM37 and PSL/PSA,38 were stand-alone, that is, were not integrated with any other tools, and supported the requirements analysis function by providing automation, either for a specific method or a group of methods. These tools consisted of hundreds of thousands of lines of code and ran on large mainframe computers. PSL/PSA used only very limited graphics in its early versions but later versions had improved graphics capabilities. SREM made heavy and effective use of graphics from the beginning. Both included some form of behavior modeling. In this same time frame, stand-alone traceability tools began to be developed.[19]

By the mid-1980s, integrated software development environments began to become available that, while they were more complex software products than the earlier tools, ran on powerful (usually UNIX-based) workstations that were smaller in size and lower in cost than mainframes. Examples include Software through Pictures,39 CADRE Teamwork, and similar products. These environments included requirements analysis tools, which generally supported one of the standard graphical analysis methods, and might also include traceability tools that enabled the requirements to be traced through design, implementation, and test. Some of the integrated environments provided a choice of analysis tools, such as tools that supported different methods. As computers continued to become smaller, more powerful, and lower in cost, full software development environments (known as computer-aided software engineering [CASE] or software engineering environments [SEE]) became available on desktop computers, and economics permitted providing such a computer to each member of the development team and networking them together.

CASE environments seemed full of promise during the late 1980s but somehow that promise has not been translated into pervasive use in the embedded systems and scientific software markets40; perhaps there was a perception that the environments supported a waterfall or baseline management model of end-to-end software development better than the prototyping, evolutionary, or incremental models that became increasingly popular. The requirements tools that are part of these integrated environments are, of course, limited to the market penetration of the environments themselves. In the application area of transaction-oriented systems such as financial and database, tools implementing an entity-relationship model such as Texas Instruments's Information Engineering Facility have revolutionized the way software is developed, but in the scientific and embedded application areas stand-alone tools, if any, are used. Perhaps the current trend toward object-oriented technologies will revitalize the market for tools and for integrated tool environments.

The role of tools and methods

In conclusion, a few words are in order about the role of tools and methods in requirements engineering, and indeed across the full scope of software and systems engineering. Proper use of tools and methods is an important part of process maturity as advocated by the Software Engineering Institute, [22] and process maturity has been shown to lead to improvements in a software development organization's productivity and quality.41 It has equally been shown that developers cannot adopt or purchase methods and tools and just throw them at the problem. Selection of tools and methods requires study of their applicability and compatibility with the organization's practices. A process must be identified that meets the organization's needs; methods must be selected that support the process, with tool availability one of the factors to be considered; then and only then should tools be purchased. Commitment, training, and the time to make the transition are essential. Tools and methods are not a panacea: if selected and used correctly they can provide great benefits; if regarded as magic solutions or otherwise misused, they will prove an expensive failure.

References

[1] Naur, P., and B. Randell, eds., Software Engineering: Report on a Conference Sponsored by the NATO Science Commission, Garmisch, Germany, 7–11 October 1968. Scientific Affairs Division, NATO, Brussels, January 1969.

[2] Alford, M. W., and J. T. Lawson, "Software Requirements Engineering Methodology (Development)." RADC-TR-79-168, U.S. Air Force Rome Air Development Center, Griffiss AFB, NY, June 1979 (DDC-AD-A073132).

[3] Schwartz, J.I., "Construction of Software, Problems and Practicalities," in *Practical Strategies for Developing Large Software Systems*, E. Horowitz, ed., Addison-Wesley, Reading, MA, 1975.

[4] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, NY, 1990.

[5] Boehm, Barry W., "A Spiral Model of Software Development and Enhancement," *Computer*, Vol. 21, No. 5, May 1988, pp. 61-72.

[6] Davis, Alan M., Edward H. Bersoff, and Edward R. Comer, “A Strategy for Comparing Alternative Software Development Life Cycle Models,” *IEEE Transactions on Software Engineering*, Vol. 14, No. 10, October 1988, pp. 1453-1461.

[7] Cameron, John R., “An Overview of JSD,” *IEEE Transactions on Software Engineering*, Vol. 12, No. 2, February 1986, pp. 222-240.

[8] Royce, Winston W., “Managing the Development of Large Software Systems,” Proceedings, IEEE Wescon, August 1970. Reprinted in *Proceedings, 9th International Conference on Software* Engineering (Monterey, CA, March 30-April 2, 1987), IEEE Computer Society Press, Washington, DC, 1987, pp. 328-338.

[9] McCracken, Daniel D., and Michael A. Jackson, “Life Cycle Concept Considered Harmful,” *ACM Software Engineering Notes*, Vol. SE-7, No. 2, 1982, pp. 29–32.

[10] Gladden, G. R., “Stop the Life Cycle, I Want to Get Off,” *ACM Software Engineering Notes*, Vol. SE-7, No. 2, April 1982, pp. 35–39.

[11] Brooks, Frederick P., Jr., Chairman, *Report of the Defense Science Board Task Force on Military Software*, Office of the Under Secretary of Defense for Acquisition, U.S. Department of Defense, Washington, DC, September 1987.

[12] Gomaa, Hassan, and D.B.H. Scott, “Prototyping as a Tool in the Specification of User Requirements,” *Proceedings, Fifth International Conference on Software Engineering (San Diego, CA, March 9-12, 1981)*, IEEE Computer Society Press, Washington, DC, 1981, pp. 333-342.

[13] Verner, June, and Graham Tate, “Third-Generation versus Fourth-Generation Software Development,” *IEEE Software*, Vol. 5, No. 4, July 1988, pp. 8-14.

[14] Cobb, R. H., “In Praise of 4GLs,” *Datamation*, July 15, 1985, pp. 36–46.

[15] Boehm, Barry W., ed., *Software Risk Management*, IEEE Computer Society Press, Los Alamitos, CA, 1989, p. 434.

[16] Fairley, Richard E., and Richard H. Thayer, “The Concept of Operations: The Bridge from Operational Requirements to Technical Specifications,” in *Software Engineering*, M. Dorfman and R.H. Thayer, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997.

[17] *Acquisition Software Development Capability Evaluation* (2 volumes), AFMC Pamphlet 63-103, Department of the Air Force, HQ Air Force Materiel Command, Wright-Patterson AFB, OH, June 15, 1994.

[18] Davis, Alan M., private communication, 1996.

[19] Dorfman, Merlin, and Richard F. Flynn, “ARTS—an Automated Requirements Traceability System,” *Journal of Systems and Software*, Vol. 4, No. 1, 1984, pp. 63-74.

[20] Palmer, James D., “Traceability,” in *Software Engineering*, M. Dorfman and R.H. Thayer, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997, pp. 266-276.

[21] Goguen, Joseph A., and Charlotte Linde, “Techniques for Requirements Elicitation,” *Proceedings of the International Symposium on Requirements Engineering (Colorado Springs, CO, April 18-22)* IEEE Computer Society Press, Los Alamitos, CA, 1993.

[22] Paulk, Mark C., et al., *Capability Maturity ModelÒ for Software, Version 1.1*, CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, February 1993. See also Paulk, Mark C., et al., *Key Practices of the Capability Maturity ModelÒ for Software, Version 1.1*, CMU/SEI-93-TR-25, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[23] Boehm, Barry W., “Verifying and Validating Software Requirements and Design Specifications,” *IEEE Software*, Vol. 1, No. 1, January 1984, pp. 75-88.

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

[25] Davis, Alan, et al., "Identifying and Measuring Quality in a Software Requirements Specification," *Proceedings of the First International Software Metrics Symposium* (Baltimore, MD, May 21-22) IEEE Computer Society Press, Los Alamitos, CA, 1993.

[26] Swartout, W., and R. Balzer, "On the Inevitable Intertwining of Specification and Design," *Communications of the ACM*, Vol. 27, No. 7, July 1982, pp. 438-440.

[27] Hatley, Derek J., and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, NY, 1987.

[28] Davis, Alan M., *Software Requirements: Objects, Functions, and States*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[29] Svoboda, Cyril P., "Tutorial on Structured Analysis," in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990.

[30] Ross, Douglas T., "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Transactions on Software Engineering*, Vol. 3, No. 1, January 1977, 16-33.

[31] Bjoerner, Dines, "On the Use of Formal Methods in Software Development," *Proceedings, 9th International Conference on Software* Engineering (Monterey, CA, March 30-April 2, 1987), IEEE Computer Society Press, Washington, DC, 1987, pp. 17-29.

[32] Norris, M., "Z (A Formal Specification Method). A Debrief Report," STARTS, National Computing Centre, Ltd., 1986. Reprinted in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990.

[33] Reilly, John R., "Entity-Relationship Approach to Data Modeling," in *Software Requirements Engineering*, 2nd ed., R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1997.

[34] Ward, Paul T., and Stephen J. Mellor, *Structured Development Techniques for Real-Time Systems* (3 vols.). Prentice-Hall, Englewood Cliffs, NJ, 1985.

[35] Hatley, Derek J., "The Use of Structured Methods in the Development of Large Software-Based Avionics Systems," AIAA Paper 84-2592, Sixth Digital Avionics Systems Conference (Baltimore, MD, December 3-6), AIAA, NY, 1984.

[36] Bailin, C.A., "Object Oriented Requirements Analysis," in *Encyclopedia of Software Engineering*, John J. Marciniak, ed., John Wiley & Sons, NY, 1994.

[37] Alford, Mack W., "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, Vol. 18, No. 4, April 1985, 36-46.

[38] Sayani, Hassan, "PSL/PSA at the Age of Fifteen: Tools for Real-Time and Non-Real-Time Analysis," in *System and Software Requirements Engineering*, R.H. Thayer and M. Dorfman, eds., IEEE Computer Society Press, Los Alamitos, CA, 1990, 403-417.

[39] Wasserman, Anthony I., and P. A. Pircher, "A Graphic, Extensible Integrated Environment for Software Development," *ACM SIGPLAN Notices*, Vol. 12, No. 1, January 1987, pp. 131-142.

[40] Lewis, Ted, "The Big Software Chill," *Computer*, Vol. 29, No. 3, March 1996, pp. 12-14.

[41] Herbsleb, James, et al., *Benefits of CMM-Based Software Process Improvement: Initial Results*, CMU/SEI-94-TR-13, Software Engineering Institute,

Carnegie Mellon University, Pittsburgh, PA, August 1994.

About the author

Merlin Dorfman is a technical consultant in the Space Systems Product Center, Lockheed Martin Missiles and Space Company, Sunnyvale, Calif. He specializes in systems engineering for software-intensive systems, software process improvement, and algorithm development for data processing systems. Merlin Dorfman has no affiliation with the Software Engineering Institute.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

Introduction

NEWS AT SEI

This article was originally published in News at SEI on: March 1, 1999

Software requirements engineering

Done well, requirements engineering presents an opportunity to reduce costs and increase

the quality of software systems. Done poorly, it could lead to a software project failure.

So it's no surprise that there has been much interest in software requirements engineering

over the past 10 years or so. Many software project failures have been attributed to

requirements engineering issues. These include poorly documented requirements,

requirements that were impossible to satisfy, requirements that failed to meet the needs of

users, and requirements creep--the gradual inclusion of unanticipated, undocumented, and

poorly considered requirements.

Even when projects do not fail outright, software developers now recognize that errors

that occur early in the development life cycle, particularly at the requirements stage, turn

out to be the most difficult and costly to fix. This is especially true when the errors are not

discovered until late in the life cycle--perhaps at implementation.

The relatively recent attention to requirements engineering is fairly typical of the patterns

seen in earlier advances in software engineering. Software engineers initially focused

on

programming methods, then on design methods, and are now focusing on requirements

methods, in an attempt to introduce more discipline in the software engineering process.

In the early days, requirements were developed in English text, but over time have

evolved into structured and in some cases formal specifications.

More recently there has been interest in requirements elicitation, because working with

non-technical people can be among the most challenging areas of software development.

In an effort to communicate with non-technical customers and understand their needs,

software developers have teamed with experts from the social sciences to gain new

perspectives on solving the thorny communication problems that can plague an

acquisition effort.

In this section

In this release of *SEI Interactive*, we explore requirements engineering from several

complementary points of view. Our Background article, “Requirements Engineering” by

Merlin Dorfman, is reprinted here by permission from the Institute of Electrical and

Electronics Engineers. First published in 1997, this article provides a high-level, tutoriallike

presentation of the field of requirements engineering. The article reviews the progress

made in requirements engineering over the past decade and characterizes different

development methodologies as they pertain to requirements.

As befits a challenge of this magnitude, every area of research at the SEI confronts some

aspect of requirements engineering, and is resulting in exciting work on a number of

fronts. You can learn more about the work briefly described here in this issue’s *Spotlight*

article, “Requirements Engineering at the SEI,” and in our “Requirements Engineering

Roundtable”:

- A significant aspect of systems engineering is the focus on tradeoff analysis. A good

example of the current synergy between systems and software is the recent work that

the SEI has done on software architecture analysis. Architecture analysis reveals how

the choice of architectural structure affects questions such as how maintainable a

system is, how well it performs, how well it scales, how secure it is, and how reliable

it is.

• Tradeoffs must also be considered when incorporating commercial off-the-shelf (COTS) products into systems. Developers and acquirers of systems must be aware of the rapid changes in commercial technology and how they affect the requirements engineering process. Department of Defense (DoD) policy strongly encourages the DoD program manager to be innovative and to embrace best commercial practice with respect to requirements. Section 2.3.1 of DoD 5000.2-R [1], Evaluation of Requirements Based on Commercial Market Potential, states

Researching the potential of the commercial marketplace to meet system performance requirements is an essential element of building a sound set of requirements. In developing system performance requirements, DoD Components shall evaluate how the desired performance requirements could reasonably be modified to facilitate the use of potential commercial or non-developmental items, components, specifications, open standards, processes, technology, and sources.

• Concerns about network security and information survivability are introducing new requirements into the process of developing systems.

• The Capability Maturity Model® Integration (CMMI) project, a joint effort of the SEI, government, and industry, takes a holistic view of the requirements engineering process and acknowledges that software and systems cannot easily be separated.

• New work at the SEI is focused on how to use technology to improve the requirements process. For example, the SEI is developing simulators to help organizations establish requirements and evaluate tradeoffs for survivable systems,

and to determine how long it will take a team to develop requirements and how well those requirements will meet a particular quality standard. The SEI's work in these areas is motivated by the growing interest in *spiral development* [2].

Finally, our Links feature offers a guided tour of information available on the Web about requirements engineering.

Notes

1. "Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs," DoD Regulation 5000.2-R, Change 2, paragraph 2.3.1. October 6, 1997. http://www.acq.osd.mil/api/asm/product.html

2. For information about how spiral development was used in the Expeditionary Force

eXperiment, (FX), an ongoing effort on the leading edge of DoD acquisition, see

http://www.af.mil/news/airman/1198/world1.htm.

The Software Engineering Institute (SEI) is a federally funded research and development

center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon

University.

SM IDEAL, Interim Profile, Personal Software Process, PSP, SCE, Team Software

Process, and TSP are service marks of Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, CERT Coordination

Center, CERT, and CMM are registered in the U.S. Patent and Trademark Office.

For more information

Contact Us

info@sei.cmu.edu

412-268-5800

Requirements Engineering Roundtable

NEWS AT SEI

Authors

Alan M. Christie

Sholom G. Cohen

David P. Gluch

Nancy R. Mead

Mark C. Paulk

Patrick R. Place

This article was originally published in News at SEI on: March 1, 1999

This article captures an exchange of ideas among members of the SEI technical staff who work on different technical initiatives:

? Alan Christie is active in promoting the application of process and collaboration technologies to make the practice of software development more effective.

? Sholom Cohen's current research activities include object technology, software product line practices, and product line introduction.

? David Gluch is investigating software engineering practices for dependably upgrading systems, focusing on software verification and testing.

? Nancy Mead, guest editor of this release of SEI Interactive, was general chair of the International Conference on Requirements Engineering last year. She

is currently involved in the study of survivable systems architectures.

? Mark Paulk was the "book boss" for Version 1.0 of the Capability Maturity

Model Ò (CMM Ò) for Software and was the project leader during the

development of CMM Version 1.1. He is also actively involved with software

engineering standards.

? Patrick Place is a member of the SEI technical staff in the COTS (commercial

off-the-shelf)-Based Systems Initiative.

The problem of requirements engineering

Bill Pollak (BP) (moderator): At the SEI, we look at requirements engineering from

the different perspectives of our technical initiatives. How can we improve requirements

engineering as part of our mission to improve software engineering practices and

technology?

Patrick Place (PP): I honestly believe that requirements are the root of all evil. A lot of

the problems that we face in our work with COTS (commercial off-the-shelf)-based

systems stem from requirements--this belief that you can do things the way you used to

do them. I just heard someone say yesterday, “We had this great success in setting

requirements and in developing this particular system.” Their “success” was that they got

the system fielded. It’s an appalling system--it’s unmaintainable, it can’t be upgraded,

and it’s going to be a disaster. But they’re ready to do the same thing again because

they’ve had a success!

You almost wish that the system had not been fielded. But I think if we want to be

software engineers, we have to consider the entire cycle, the entire life of our systems

from inception to final turnoff, cradle to grave. And that involves figuring out what that

system should be, which is collecting and eliciting requirements and maintaining those

requirements throughout the life of the system. Because if you just do it once and then

throw it away and let the system evolve, you’re in trouble.

Sholom Cohen (SC): In the case of our work with product lines and architectures,

customers either have nothing in the way of product line requirements and are looking to

establish some basic set of requirements--they aren’t really looking in detail at how to

elicit, manage, or track requirements--or they come to us with an architecture already in

hand and say, “Now what do we do with it?” Or they come to us with a system that’s in

trouble and say, “How do we reengineer it?” So it doesn’t seem like people are coming to

us saying that they need help with managing requirements. We might say that there’s a

problem there because they can’t track the requirements. And they say, “Well, give me a

tool to do it.”

Nancy Mead (NM): We see a lot of people coming to the CERTÒ Coordination Center

and asking us to analyze their existing systems to see where the security holes are. Right

now we’re really trying to push the idea of having them look at their architectures up

front to see if they’re survivable. There is not as much awareness of these up-front

survivability issues as there should be, but the awareness is growing.

Alan Christie (AC): Often what is required in developing a set of requirements is a

broader systems perspective--to be able to see the whole as more than the sum of the

parts, to be able to assess the behavior of the system as an integrated entity. Complex

feedback effects are very difficult to understand and often counter intuitive. As system

complexity increases, it is more and more difficult to predict, at the requirements phase,

what exactly will happen. As a result there is requirements churning, delaying project

deadlines and pushing up costs.

I believe this situation can be addressed with the appropriate use of simulation

techniques. Simulation has been used for many years within the engineering community

to predict everything from traffic flow to aircraft wing loadings to the likelihood of a

reactor accident. However, the software community has, ironically, been slow in adopting

this technology. I don’t believe that simulation is a silver bullet, but it is more than a

technology in the sense that its use can raise awareness with respect to the importance of

dynamic effects in complex software systems. By using simulation to support

requirements, a customer can achieve a much-improved sense that a supplier understands

the issues. In addition, if changes are proposed, a simulation of the changes is likely to

wring out potentially unforeseen consequences.

Simulation software is now mature and readily available, but it will probably take a few

dramatic examples of simulation supporting software requirements before the community

takes notice.

PP: A lot of our activities in the COTS-Based Systems Program have been based on a

customer coming in and saying, "I've got this legacy system, it's ancient, we have to

build a new one, how do we do it?" And the first thing is for them to even understand

what it is they have in their existing system. They have documents that are hundreds of

pages long that are supposed to describe their requirements, but they don't. They're a

mishmash of operational procedure, of detail, of flowcharts...but the documents don't

provide any understanding of what their requirements are.

NM: And they're telling you that they really have no choice; they're up against a wall.

PP: Some are not quite as up against the wall as others. But they're all under a federal

mandate to use COTS products, one way or the other. And so they're trying to figure out

how to do it. We tend to work on the design and the technology, but one of their big

problems is understanding the requirements that they need to build to.

NM: The other problem in the requirements area is that requirements are higher risk than

some of the downstream problems we deal with, and we don't have nice, neat solutions to

hand to people for problems related to requirements. We can't say "If you go off and use

this analysis technique or that elicitation technique, everything is going to be fine."

The human side of requirements engineering: natural language vs. formal methods

BP: Requirements are more than a technical problem; in large part, they are a people

problem too.

Mark Paulk (MP): Most of the real requirements engineering value is in the people

stuff. It's fundamental communications, not writing things down in a mathematical

formula. It's going out there and talking to people, and eliciting what the requirements

are, then capturing them in some way that an MBA can understand.

Probably the best book I know of in the area is Gause and Weinberg's *Exploring*

Requirements [Gause 89]. They have exercises where you look at a sentence like "Mary

had a little lamb," and you see that the meaning changes depending upon which word you

emphasize. So we have to look at the whole idea of the way that we communicate, which

is really an interpersonal skills issue.

There are folks out there who are doing good work. Barry Boehm, for example, with his

win-win stuff [Boehm 99], is in my opinion one of the folks who is at the forefront of this

kind of activity. All of us have to do requirements elicitation and analysis for our

particular projects. But in terms of looking at requirements engineering as a discipline, I

think you really need somebody or a group of people who focus more on the elicitation

side and the capturing of requirements, which is a communications issue more than it is a

formality issue.

PP: I'd like to take issue with one of your points. I agree that requirements are a people

issue, but you jump to the conclusion that writing things down mathematically is not the

right thing to do; I think this is a bit of a leap. I was involved in an exercise with the IEEE

POSIX.21 standard for realtime distributed systems communication. We communicated

in English to the people who were developing the requirements about the flaws that they

had in their existing requirements set, but that communication was based on a

mathematical analysis. I agree with you wholeheartedly that I would not take 20 pages of

mathematics--or even one page--to the average system acquisitions individual and say,

"Here, now you know what the system is." But you can make the interpretation back into

the system's own language or domain of discourse because you can see from the

mathematical logic the consequences of your decisions; I will tell you what those

consequences are not in a stream of symbols, but in a stream of words that have meaning

to you, and you can tell me if I'm right or wrong. And if I'm wrong, I can adjust my

model, and if I'm right, you can adjust your requirements.

MP: I agree, you're absolutely right. Those kinds of analytical techniques are very useful.

The point I was making is that a lot of the work I see that is reported under the

requirements engineering label is really dealing with a requirements specification

language, a formal methods kind of approach: "We're going to have a complete,

consistent, unambiguous set of requirements." And then you go out and you try and

explain that, because when you try to translate from the formal specification back to the

English, you inject ambiguity back in. That's the reason you went to the formal

specification in the first place.

You can get fairly detailed with a rigorous analysis, but when you try to abstract that

back up to the CEO or the MBA or whomever your customer is, things you have captured

absolutely precisely are just not what they really meant. Or as we've said several times,

they don't know what they really mean, but they'll figure it out as we iterate with them

on what the system should be.

David Gluch (DG): One of the things we're talking about in model-based verification is

to use formalism and pragmatic, focused models as communication/understanding

vehicles. As people analyze requirements specifications, they create an abstracted model,

and there is communication. They begin to understand the engineer who is building the

system, and they also make users, clients, and other stakeholders aware of the issues that

emerge. So we see that using formal language effectively as a communication as well as

an analysis vehicle is a very effective way to identify errors in requirements and make

them explicit. People may consciously decide about a certain requirement, for example,

"I'm not sure about this, I'd like to defer it."

MP: Is this coupled with tools?

DG: It can be, in terms of the actual checking part of the model, but it's mostly just the

process of building the model--of abstracting out the essential elements--that helps focus

discussion about a system and also identifies ambiguities.

PP: That's a communication from the people who have the requirements in their heads to

you the engineer. It doesn't address the communication from you the engineer back to the

people so that you can fix what's in their heads because they've got it wrong.

MP: Let's broaden what we said earlier, which is, if you know who your audience is,

then you can communicate in a way that they will understand. If you're talking to a

software programmer, and you say, "Now here is the language specification for C++,"

you might use the formal BNF specification to help the programmer understand the

syntax for the language. But if I were trying to teach some manager how to do

Programming 101 so he could write a little sort program, and I put one of those

specifications down, I can predict that there would be some resistance to that form of

communication.

PP: A course at the University of Manchester actually did that. They gave first-year graduate students a formal specification, plus a natural-language discussion, for their first exercise. In fact, for all of their programming careers, students used formal specifications in their exercises. And the university found that successful--that's why they did it.

MP: I made that mistake too when I was starting out teaching. It didn't work out as well for me. I guess that natural-language discussion is critical: the better you are at that, the more success you're going to have.

PP: Yes, you need the two together.

NM: I think that's a problem that we see a lot. We do need both, and people don't want to invest in both. They want to pick one or the other, and I'm not sure that one or the other can do the job that needs to be done.

AC: I don't think that there is any real conflict between natural language and formal requirements. Usually formal requirements are applied to narrowly focused elements of the problem while natural-language requirements complement this in the broader scope.

For high-reliability and safety-critical systems, formal requirements make a lot of sense for precisely describing the system's interactions, but that's not necessarily best for providing an understanding of overall capability.

PP: The most important part of a formal specification is the natural language that surrounds it. Any mathematical model can state very precisely what the requirements are, but there's no intuition. And you can provide all the intuition about that precision in natural language, so that it's not just 20 pages of mathematics. It's mathematics plus text that says what the mathematics mean.

Requirements engineering in other disciplines: Is software unique?

BP: Is requirements engineering particularly difficult with software?

DG: We often seem to function in isolation, but formal representations are routinely used in structural engineering. They go to mathematics very quickly, they go to formal representations, and they're integrated--mathematics are part of how they define requirements, part of how they accomplish this communication and translation to the

engineering staff and to the other people involved. I don't think there is anything fundamentally different about building software than building anything else, but there's a larger focus on intellectual intangibles up front that are more difficult to describe.

PP: The issue is one of complexity. We are creating software with thousands of functions, many times more complicated than any other manufactured artifacts. Even well-understood programs have incredible complexity. In a previous roundtable discussion in *SEI Interactive* (http://interactive.sei.cmu.edu/Features/1998/June/COTS_Roundtable/Cots_Roundtable.htm), David Carney likened COTS software to bridges as opposed to screws; however, I think that he's doing software an injustice--it's more like building a country's road system when the pieces we have to interconnect are the towns! This complexity takes software out of the realm of the other engineering disciplines.

A second significant difference is that software is, by its nature, of extreme generality. A toaster makes toast and could, if you bought an expensive enough toaster, be coupled to all sorts of other household artifacts so that your morning toast was available, say, 10 minutes after your morning shower--whenever that occurred. That said, it's still a toaster, and if I wished I could create the mathematical language to describe and engineer toasters.

The mathematics of bridges are well understood and relatively simple. However, for software, since it has no defined purpose, the best we can do is create a mathematics capable of modeling any aspect of software. Unfortunately, this leads us to all computable functions (a technical term, not to be confused with computation)--this is a lot of mathematics to cover and means that we are without a domain-specific language to use for our engineering (unlike bridges that deal in tensile strength as well as forces and moments).

Another point to bear in mind is that software development is really new; we've only been developing what we consider to be software for a tiny fraction of the time during which we've been building bridges.

AC: One distinction between most other engineering disciplines and software is the fact

that, in traditional engineering, the results of what you produce are mostly tangible. Buildings, bridges, and aircraft all have a physical presence that software does not. I think this physical presence allows one to more easily visualize the issues and to think in metaphors. For example if I understand the reason why bridges are structurally sound, I can probably use that knowledge in designing safe buildings--or even aircraft. I'm not sure that this intellectual reuse is as portable with software. At least it is not being used that way today.

DG: My comment relating to specifications for other disciplines was intended to be a very general observation that other engineering disciplines rely heavily on mathematics as a basis for describing and understanding their systems. I did not mean to select structural engineering specifically. I believe this is true across the board in engineering endeavors.

My point is simply that other engineering disciplines rely on mathematics (formalism) and so, if software engineering is to be an engineering discipline, it too should include mathematical formalism as integral to doing “good” engineering for software systems. The issue is not whether formalism should be included but rather how it should become part of the discipline.

Also, I was speculating on what differentiates software from the other engineering disciplines. While it may seem that software engineering does not have much in common with mechanical engineering, I think it does. But consider another example--electrical engineers (digital) designing microprocessors or custom application-specific integrated circuits. I believe the issues and problems here closely align with those of software implementation. It can be argued, though, that the similarity is at the design/implementation levels rather than at requirements. But in looking at requirements, one may ask how software requirements engineering differs from systems requirements engineering.

Pat, relating to the complexity premise, I am not sure about this perspective. Consider the Boeing 777. I believe that the complexity of the entire aircraft is greater than the complexity of the software that comprises only part of it, and that the requirements

specification for the software can be considered as resulting from doing the system design for the aircraft.

MP: I agree with Alan that it is easier to have a set of shared paradigms when you have something tangible to look at. So when you're saying "I want to build a bridge," if I'm the mayor of Pittsburgh, I can say "I want a bridge to go from here to over there and I want to be able to run 50,000 cars every day over it." And so you can specify from the user's perspective.

Dave is absolutely right, when the engineers started doing that, they started getting into the mathematics from Day 1, but those mathematics are communicated to the city engineering office, they're not communicated to the mayor. There are different audiences for the different ways that you capture things, and the problem that we run into in the software world is not that we don't have different audiences, it is that we try to communicate the same way to all of them.

Communication

NM: In one of the projects that I was on, we had a series of design reviews. And one of the later design reviews was a review of the user interface and displays. At that review, we found out that these folks who had sat through all the previous reviews had no understanding of what was going on, because they were looking for data on the displays that wasn't being computed in the system; and if they had understood everything that had come before then, they would have realized that not only was the data not being displayed, it wasn't there, it didn't exist. Because they didn't understand the methods that we were using to present our designs, they were unable to internalize that until they actually saw the user interface; that was the only way they could comprehend what it was that we were trying to do.

MP: That's why you see such a big emphasis on use cases these days. Operational scenarios are popular because it's in the actual use that people internalize what it is that they're getting. Rapid prototyping is another technique--all of these techniques are ways of getting the user wrapped up into the system.

DG: Achieving this balance is a key problem--keeping it in the environment. Any

descriptions of requirements that evolve should be characterized in terms of the environment so that they become more real. The entities that are identified need not be tangible, but they must be real in terms of the user, and I think it's important to keep a balance between what's real on the one hand and the need to consider COTS products on the other. Because COTS products will constrain the designer and ultimately perhaps the performance of the system. This is not explicitly communication, but it's really wrestling with ideas that have traditionally been discussed in terms of the "what" and the "why." Use cases, scenarios, and other similar strategies enable people to describe the system and how it works within the environment.

SC: Use cases came up for us recently. We had someone come in to talk about them. This person's organization had developed, I think, 156 use cases to describe its system. The use cases were stated in terms of processes that the system had to go through. We were sitting there with the person who had developed all the requirements, and also across the table were all of the program managers. He said "What I would like to do is hit on one of these functions and see what use cases apply." And the guy's jaw kind of dropped, because there was no mapping between the use cases, which were process oriented--how to do tracking, analysis, engagement planning--and the functions that were actually built into the system.

AC: From an organizational perspective, it's increasingly common for a customer for software and the supplier of that software to be geographically distributed. This usually means that they have to get together on a periodic basis to "sync up." During the formative stages of a project, when requirements are being developed, this interaction can be particularly critical, and interactions should be most frequent. However, the impediment of distance means that such interactions do not take place as frequently as they should, with the result that communications quality can be degraded. Phone and email communications just do not measure up when high bandwidth interactions are required. Clearly the quality of the requirements lays a foundation for the quality of the resulting software system, so there should be a great motivation to really work hard at effective means to communicate easily early during the early stages of such projects. This is why I believe that collaboration technologies should be used to help groups develop requirements when these groups are geographically separated. These

technologies are rapidly maturing and coming down in cost, so it is becoming

increasingly appealing to use them. They allow for a wide variety of types of interactions

among individuals, and the automatic capture of session interactions provides a historical

record of activities.

What are the "real" requirements for the system?

BP: It seems that requirements can exist at different levels of granularity. How can

software engineers manage these different levels and determine the "real" requirements

of their systems?

PP: Let's take the FAA system as an example. The FAA has a CD-ROM full of

documents, that they call "functional specifications," and these are all in terms of user

functionality, such as "The user types these two letters and these two digits, and this is

what will happen." So in essence, they are the requirements for the system, and that

document is maintained and is kept up to date. And, given the life-threatening nature of

their systems, people do not make changes without good reason. But is that document

really a "requirements document?" My guess is that it is not, at least at the level of "What

are the requirements of the system?"

The requirements of the system are that it be able to maintain separation of aircraft, to

maintain the safety of people as they're flying en route and as they're landing; it's not

that the controllers shall be able to type these two letters and these two digits and achieve

this effect. I think that people lose sight of the real requirements, and they don't see the

value of investing the effort in maintaining currency between whatever the requirements

document is and the code. Because while the design documentation and the architecture

documentation may never change, they know they're going to change the code. So there

is a tendency to view everything else as overhead. That's why you get the situation that

we've all seen with systems that evolve, where the initial requirements document

becomes irrelevant. We do not recognize that the requirements are recorded in a living

document that must be maintained and that must survive and evolve throughout the life of

the system. We just don't do that, because it's expensive and there's no return on it; it

doesn't get another byte of code written.

NM: One interesting example is the use of the Patriot systems during the Gulf War. By keeping the system in operation much longer than was intended, accuracy was lost, and hence the anti-missiles didn't go in the right direction. The change was a procedural one-- just take it down and re-initialize it periodically.

PP: Early versions of IBM's AIX operating system had a memory fragmentation problem. After a couple of weeks of continual operation, they'd run out of virtual memory. Procedurally, you needed to reboot the system. But if you're running on a 24-

hour-a-day, 7-day-a-week system, you do not want to reboot that system. Yes, you can get around these things, but I see confusion in the fact that people have these operational procedures documents but don't realize that there are also requirements on the overall system. We think of requirements that are system requirements or software requirements, but we don't consider that the man in the loop and other environmental factors really form this thing that is the system.

MP: Let me ask a question that builds on what you and Nancy have been saying. Should the user manual or the operator's manual be considered the real requirements specification for the system? For example, in the case of the FAA and the need for separation of aircraft, a requirement might be that there should be an alarm that goes off or some kind of blinking light that says these craft are getting too close together. Or the procedure for operating the system should say that when these two aircraft get too close together you should tell them to go in different directions or something. I know that there are non-technical requirements in there, but one of the things that I've encouraged folks to do is to make sure that their user manuals and what the code actually does are consistent with one another.

PP: But this still misses the point. Doing that gets you into the world of "We have 2000 requirements, and X number of 'shalls' in our document," and this is what you have to address. At some level, yes, you have to have that, that forms a contract. But is that something that's manageable or maintainable, or even something that I can elicit from

people?

MP: Let me rephrase the question. Should anything be considered a real requirement that is, in essence, invisible to the user? A lot of times when we get into functional requirements, they're actually design statements, they're not really requirements. It's the requirements about what the system will do, what the user can expect to achieve, which I would hope would be captured in some kind of user manual or something like that.

PP: But there are cases where it is important to require things that may be invisible to the user, that may impinge upon the design. For example, if the FAA writes requirements only in terms of things that are visible to the user, they will have many different types of maintenance to do. If they can require that everybody runs on this operating system or uses that hardware, which is completely invisible to the user, they can save themselves a whole bunch of maintenance.

MP: I would characterize that as being part of the maintenance manual. Depending on what environment you're from, a lot of times there's a maintenance document that is part of the deliverable set, which includes the user manual, the operations manual, the maintenance manual...

PP: Yes, but in the world of COTS, you don't have a maintenance manual. You don't maintain the operating system. Periodically you will have to do operating system upgrades, but you don't maintain the operating system. I think that you would rather have one operating system and one set of procedures for how to reinstall that operating system than to have 20.

NM: I think that the place where this breaks down is that, again to use the FAA example, all of the operational procedures tell you what to do with aircraft with the assumption that you're tracking them correctly. But there's really a requirement of the system that you track all aircraft, and that might not appear in the user manual. And all of the software that's underneath it isn't captured in the statement that you're going to display.

PP: The user manual is all about how and what you can do with the information that you have available. But there are other requirements for completeness and currency of that information that probably don't appear.

Documenting requirements and maintaining architectural integrity

DG: If I could put a little different spin on this, one of the key areas I see is getting the thing right. Many if not most of the errors that occur downstream, even in code, were requirements errors. So there are two elements of getting requirements right. One is something we've been talking about--eliciting them and capturing the real needs. Once that's accomplished and we have in fact described real needs, are they consistent and correct in and of themselves? That's the kind of analysis that we're focusing on. Are they complete, consistent, do they make sense, do they really accomplish the objectives that the stakeholders wanted? These questions must be addressed at the requirements level, especially given the data that shows that the cost of correcting an error that occurs downstream that was originally injected in the requirements is quite significant.

MP: How do you ensure that you keep a consistent record of what the requirements are? We've all observed that, over time, the code changes, the design changes, and the requirements change. So the point that Dave is making is absolutely valid, but the problem is that if we had a stable set of requirements that was unchanging, then the more rigorous techniques would be much easier to use. But when you have a dynamically changing environment, how do you deal with the consistency and make sure that everything is kept up to date? It's a human behavior issue as much as anything else.

DG: Right, and that's true regardless of the language that's used, whether it's formal language or natural language. But a potential solution is to capture as many of the requirements as appropriate and can be effectively accomplished in a formal language and then allow automation to handle a lot of the application-independent things that are

problematic and that require a lot of time but not a lot of intellectual activity on the part of humans. There are solutions that are tending in that direction, so that you have a line from requirements all the way through to code, and the ability to then do checking and analysis.

SC: At conferences that I have attended in the past couple of years, the emphasis has not been on the requirements phase. Even when they talk about analysis, they're talking about something that is much closer to design than it is to requirements analysis. The approach starts with the assumption that the requirements already exist, and elicitation is

not the emphasis.

In the product line area, we tend to apply object-oriented analysis to the up-front

elicitation; but again, it's in a rather narrowly focused area within a bigger problem. And

once we give our results back to the client, there's a tendency to look at architecture

development activities, and then never to return back to the requirements as they exist in

that up-front analysis effort. The requirements will be evolved through the architectures,

through change scenarios, or some other approach, which lets clients explore what

they've got in hand, what kinds of changes the system is likely to undergo in the future,

and to refine with those things in mind. They're not going to go back and elicit further

requirements through a formal elicitation or management process. And I think that's

likely to cause a problem downstream. At the end of implementation or even design, the

clients don't really know what requirements they have.

This is similar to what Pat said earlier--a lot of people come in with these legacy systems

and ask "What do we have?" So one of the things we can take back to other customers is

to say, our experience has shown that, without formally tracking and managing

requirements effectively, you're going to run into some of the same problems

downstream as those you've encountered in the past. They may hit later than they do

now, but they're still going to hit, and they're going to probably be as severe if not more

so, because systems are bigger and more complex, and they affect more people than the

old standalone systems used to.

The situation we see commonly is that people do some up-front analysis, and they have

some elicitation and some requirements, but as they get into the architectural design, they

never go back and update the requirements. So within months after the first set of

requirements is complete, the requirements have already been overcome by the

architecture, and they are never re-examined. To these people, requirements elicitation

and management aren't even issues. The issues now are management and evolution of the

architecture. There may have been some requirements notions up front, but now they're

embodied in the implementation.

NM: I think that what happens is that, over time, people lose intellectual control, and they lose the sense of integrity of the architecture, and going back further, the set of requirements, and they start changing things in ways that they don't reflect on. They don't go back and say "Does this match the original vision for this?" Sometimes the vision isn't documented or expressed really well, and so you don't really know what the impact is. But many times, people just don't think about it; they think about the immediate problem at hand and not the impact that it has on the original vision of the system, which is why you see systems that become a mishmash even if they had a pretty decent set of requirements or a decent architecture to begin with.

Improving requirements engineering

BP: We've had a very rich discussion about the problems associated with requirements engineering. In the time remaining, I'd like us to focus on some of the solutions in the various domains that you represent. Nancy, you mentioned an analysis method in the security area.

NM: In the area of survivable systems, we are developing an analysis method that we call the Survivable Network Analysis (SNA) method for software architectures. The SNA method is causing us to think about requirements for survivable systems. We find that most clients have not given a lot of thought to how to get the essential functions of their systems to survive an attack or a break-in and to continue to operate--how to recognize such an attack and how to recover from it. We call this the "three Rs": resistance, recognition, and recovery. For readers who are interested, we wrote up one of our survivable network analysis case studies (http://www.sei.cmu.edu/publications/articles/ellison-survivable-systems/ellisonsurvivable-systems.html).

We're finding that in doing architectural analysis, we are stepping back and making recommendations for requirements changes as well as recommendations for survivability.

We think there is a lot of work that needs to be done here. Many users, from defense to finance, just haven't given that much thought to the vulnerabilities that will occur when they take their systems and distribute them over a network.

One of the things we're trying to do this year in our research is to think about intrusion scenarios that we develop and work with a client on. We'd like to have a canonical or

relatively complete set of possible intrusions at the architecture level. In our initial work

with this, we basically used our own intellect to figure out what the typical types of

intrusions might be, but we'd really like to reduce that to something that could be used in

a more analytical way, so the customers could feel that they had taken into account most

of the bad things that could occur. So our focus this year is in part on that and in part on

taking into account risk analysis, because intrusions are not your normal occurrence,

they're at the far end of the bell curve of activities relative to the systems. And in

conjunction with that, we're looking at formalisms that support the work we're doing.

BP: In the sense that Pat and Dave have spoken of?

NM: There's certainly some relationship, yes.

AC: I mentioned this before but I think it's worth mentioning again. I don't believe

behavioral properties of software systems are being addressed early enough. Too many

systems have been built that don't address performance issues at an early stage, and

consequently they get wiped out during operational tests. Likewise, complex system may

not respond as intended because of unforeseen dependencies between loosely coupled

components. I think simulation can help here, and in fact, a simulation model can become

an integral part of the system's specification. It is almost certain that, after the

requirements development phase is completed, requirements will change. Such

simulation models can thus be used to investigate and address new issues as they arise--

before they become significant problems.

SC: We've found that a lot of the people we're working with are taking the scenario

approach or use-case approach. Many people are adopting this approach to move toward

object-oriented development. But we still come back to the fact that they get to a certain

point with the use cases and then kind of abandon further evolution and maintenance of

the requirements. Instead, the class hierarchy or class structure, whatever the next stage

is, is maintained because it's closer to the code or the implementation. It seems like there

are instances though, where requirements are placed under very tight control, which

would work if the requirements were true requirements. Often they degenerate into functional descriptions, and then you're really just writing in text what's happening in the design--the example mentioned earlier of hitting these two keys and having this result. Those are not really requirements, they're a description of what's going on the code.

PP: I don't think we have solutions in the COTS domain. We have, at best, advice, which is perhaps little more than a collection of things that you should know anyway. One of the more innovative approaches that we've seen is to get recognized consultants from the industry--from the marketplace--to help set the requirements for a particular product for a new system to be developed. That is, instead of getting only experts in the system to be developed, you also get marketplace leaders who can say, "If you go down this route, you will never be able to acquire anything that's COTS based." We are beginning to collect these sorts of successful approaches together. But are they repeatable? Who knows?

BP: Maybe these are sort of "best practices."

PP: Or suggested practices. Things like setting the levels of stretch of the requirements is a real problem. We see problems where you go into so much detail that you don't admit COTS solutions. That may be what you need to do. On the other hand, we've seen cases where to ensure the use of COTS-based systems, people set the requirements so abstractly that they get completely unsuitable systems offered in bids, and they have no way to reject them because they meet the stated requirements. There was no way to say that this one, which was the lowest cost alternative, cannot be employed, because its user interface, for example--which we didn't mention in our requirements--is completely unsuitable for our users, and they will reject it, and it will never get installed. So we're at that sort of level--we have advice, but not solutions.

With regard to engineering business processes, sometimes it's much easier to change the way you do business than it is to acquire a system that does the business that you're in. We saw one case where the DoD installation said, "It's much easier to adopt Oracle Financial and get that approved within the Air Force as a reporting mechanism for financial information than it is to try and acquire a system that reports on the existing reports that we're producing. But again, this is mostly anecdotal; there's no real

"practice."

We talked earlier about the POSIX work. One of the things that we did with POSIX.21 was that, when the requirements were still in the formative stage, we did an analysis of the requirements and formalized them, and then reported the problems that arose from the formalization back to the requirements group and changed the requirements because of the problems that arose from the formalization. So this was a real requirements engineering elicitation. The entire working group believed then and still believes now that this was a valuable exercise and a really great way to home in on the requirements and get higher quality requirements faster than they could have gotten without them. Again, this is another piece of anecdotal experience.

SC: We did a case study about three years ago of CelsiusTech, a Swedish firm. They have an elaborate requirements database that they've maintained. It has allowed them to work with new customers to say, "Given this set of requirements that we can support in our architecture, how do you envision your systems?" So the new system requirements are developed in light of this requirements database that's been proven over 10 years or so in implemented systems. That's one instance of a managed, tracked, effective database of requirements.

BP: Is the requirements aspect specifically discussed in the CelsiusTech technical report [Brownsword 96]?

SC: Yes, but the report is a high-level overview, so the information about the database is probably only a paragraph or two. The counter example to that are things that come up in our architecture analyses for applying the Architecture Tradeoff Analysis Method (ATAM) or Software Architecture Analysis Method (SAAM). You get stakeholders together sometimes who haven't worked together, and we're taking an existing architecture and eliciting scenarios about how the system will change, how it will be used, and how it's envisioned. And new requirements pop up, or requirements that exist that were unknown to other people in the room. So that's an example of requirements that weren't developed, tracked, or fully understood, and now that the system has been built,

things are suddenly coming to light. Again, there's no recommendation for how this could have been done more effectively, but at least bringing the stakeholders together and working through these kinds of analyses are helpful for bringing these things out early in the development--anything you can do to avoid sudden recognition when the system is being tested, as in Nancy's case of the people who didn't understand the system until they looked for data on the screen. So we hope that the architecture analysis brings the stakeholders together to recognize problems with requirements up front when the system is being designed.

So we encourage people to examine and consider requirements up front, and at least if there's a core requirement that's being used to define the evolution of the system, maintain that core even if the other details aren't maintained. If analysis turns up new requirements, we have to pull those back into this core. But I don't think we'll ever get to the stage where in the majority of cases, requirements are fully defined and explored as the system evolves.

DG: I think that what we really have been talking about here is making requirements engineering part of our systems engineering activities. That's really the essence of all of this. A lot of the problems we've identified are simply a failure to be engineering-like. We need a disciplined tracking practice as well as a technically sound approach to capturing and analyzing requirements.

PP: I agree. Engineering discipline takes a lot of time to develop—I just read Petroski's *To Engineer is Human* [Petroski 85] to understand the millennia spent "hacking" bridges, and everything else for that matter. So, it's not a surprise that software isn't engineered in the same way as the products of other disciplines. But that doesn't mean we shouldn't try and put it on a sounder footing.

References

[Boehm 99] Boehm, Barry & Port, Dan. "Escaping the Software Tar Pit: Model Clashes and How to Avoid Them." *Association for Computing Machinery (ACM) Software Engineering Notes 23*, 1 (January 1999): 36-48.

[Brownsword 96] Brownsword, Lisa & Clements, Paul. *A Case Study in Successful*

Product Line Management (CMU/SEI-96-TR-016, ADA 315802). Pittsburgh, Pa.:

Software Engineering Institute, Carnegie Mellon University, 1996. Available at

http://www.sei.cmu.edu/publications/documents/96.reports/96.tr.016.html

[Gause 89] Gause, Donald C. & Weinberg, Gerald M. *Exploring Requirements: Quality*

Before Design. New York, NY: Dorset House Pub., 1989.

[Petroski 85] Petroski, Henry. *To Engineer is Human: The Role of Failure in Successful*

Design. New York, NY: St. Martin's Press, 1985.

SEI Interactive, 03/99 page 19

http://www.sei.cmu.edu/interactive/Features/1999/March/Roundtable/Roundtable.mar99.htm

The views expressed in this article are the participants' only and do not represent directly

or imply any official position or view of the Software Engineering Institute or Carnegie

Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development

center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon

University.

An Example of Erratic Effort Estimation

NEWS AT SEI

Author

Scott R. Tilley (Florida Institute of Technoloy)

This article was originally published in News at SEI on: March 1, 1999

What role do requirements play in effort estimation for net- centric applications? Are they more important in the net-centric domain? Less important? The same? This article provides an example of erratic effort estimation that can be traced, in part, to inadequate requirements engineering.

It is accepted that requirements play an essential role in the software engineering life cycle. One of the thorniest aspects of program development is getting both the customers and the developers to understand what the other means. Requirements engineering is an area of great interest to industry, yet it receives relatively little emphasis in most academic institutions, especially with regard to the software engineering curriculum.

This article provides an example of erratic effort estimation that can be traced, in part, to inadequate requirements engineering. The context for the example is an undergraduate course on software engineering. For most students, this is their first introduction to the basic tenets of software engineering, which include requirements engineering as it influences subsequent effort estimation.

The students were asked to estimate the effort required to implement several proposed changes to a Web-site analyzer program. The Web-site analyzer was used throughout the course as a framework for group projects. This type of net-centric application was relatively new to them, which added to the challenge of producing accurate estimates.

The Web-site analyzer program

The course project was a mix of programming, testing, documenting, management, and other activities that are typical of a real-world software engineering effort. The project was to create a Web-site analyzer toolkit called “WSA.” The WSA application was created by enhancing an existing Web-log analyzer program, which was written in

C and contains about 5,000 lines of code. It uses a graphics library package for creating GIF files that form the heart of the graphical reports viewable with a browser.

The course project was meant to closely resemble a realistic software engineering exercise. The students were told to imagine that they had been hired at a new software startup company. The company had just bought out a rival company because the rival hada great Web-site analyzer program, and the market was booming for that type of application. As a new employee, the student was put on a team that was given the responsibility of enhancing the newly acquired Web-site analyzer. Students were to imagine that their boss had said that WSA needed many new features to make it more competitive in the marketplace.

Some of the boss's requested functionality could be found in other software packages, but the team's budget was nonexistent for capital purchases. Moreover, there were fewer than 10 weeks to ship the product (coincidentally, the length of the academic quarter), and going through the formal procurement process would take longer than 10 weeks. The team members were left with just one choice: they had to learn how the current Web-site analyzer program worked and enhance it. The students were warned to expect that some of the desired features would change over the 10-week period, and that their boss was fickle. In other words, they were told to expect requirements creep.

What the students know

Before the students were asked to provide effort estimates for proposed changes, they had been taught about the different types of software maintenance. However, they had been exposed to very little in the way of formal cost-estimation techniques. For example, they had not yet learned about function points as a means of measuring programmer productivity. They were asked to use the more accessible lines-of-code measure, even with its many limitations.

This was by design. The goal was for the students to use their own insights and methods for performing their first effort estimation. They would then learn more disciplined techniques, such as algorithmic cost modeling, and in subsequent exercises would compare the effort estimates arrived at using both methods.

The effort-estimation exercise

As a team, the students were asked to complete a change request form (CRF) for five different proposed changes. Part of the CRF was focused on estimating the effort required to carry out the change. Effort in this context included the number of lines of commented source code required to be changed, the number of hours required to implement the change, the number of people required, and any other information that the students deemed to be relevant. They were also asked to characterize each change as corrective, adaptive, or perfective maintenance.

To put the effort-estimation exercise in the context of their hypothetical company, the students were told to think of their group as a consulting firm bidding on a lucrative contract. On the one hand, they would want to provide an accurate bid that was low enough for their firm to be awarded the contract. On the other hand, they did not want to come in with too low a bid, because the contract would be awarded for a fixed price. That meant the deadline for finishing all change requests was fixed. Their firm would pay for any work required past the deadline, which was something they obviously wanted to avoid and to which they could relate.

The students were repeatedly told that it was in their best interest to try to be as accurate as possible with their estimates. The next assignment focused on implementing some of the changes, and potentially others not listed as one of the five requests. However, they did not know at the time which changes would be implemented.
Some of the change requests were worded in an intentionally vague manner. For example, one request said simply, “Port WSA from Linux to Windows NT.” Others provided more detail. One request even related to something they had already implemented in the previous assignment. In that case, it was more of a post-change effort analysis than effort estimation. The change request that they had to implement for their next assignment was worded as follows:

Add a graphical display to WSA that represents the information in the Web logs. For example, display each HTML page accessed, the accessing site, and the referring site (if available) as a series of nodes and edges. Assume the graphical display package is a prebuilt component that you just need to integrate with WSA.

At this point, the students did not know anything about the graphical display package that they had to integrate into their version of WSA, except that it would be provided to them as a working component. They did understand the structure of Web logs, since it was the focus of the implementation effort on their previous assignment. The estimations from the different groups on this change request made for very interesting reading.

The effort estimates

Perhaps not surprisingly, the seven student groups provided effort estimations that were all over the map. Most groups did not appear to understand what the requirements meant, and therefore their estimates were based on inaccuracies from the start. The students tended to give the change a low priority relative to the other change requests. This was
yet another instance of not being able to predict what the customer really wants!

Being typical engineering students with heavy class loads, they tended to perform their effort-estimation exercise the night before the assignment was due, leaving little time for careful thought and group discussion. However, in that respect, they were not all that different from professional software engineers who are often asked to provide an on-the- spot effort estimate based on ill-defined requirements and unknown project parameters.

The effort estimation included the number of lines of commented source code required to be changed, the number of hours required to implement the change, and the number of people required. Groups summarized the change using words such as “easy” and “long- term process,” with many adjectives in between. They estimated the number of lines of code that would have to be changed in WSA to be from 10 to 300. As the “boss” of their company, I know that a considerably higher amount of source code would need to be changed and added. However, underestimating the impact of a change request is very common, even for experienced programmers. They estimated that one, two, or three people would be required—good thing the groups had no more than four people.

The most surprising degree of variation was their time estimates. The amount of estimated time required for implementing the change varied from 2 hours to 24 hours to 120 hours to 8 days. Quite a wide range!

Discussion

Why were the students’ estimates so varied? Since they were juniors and seniors, they did have some experience with programming projects, but very little experience working in teams. Their knowledge of real-world programming projects was also quite limited.

The application domain of WSA definitely played a role, since this type of net-centric application area was unfamiliar to them. Although the format of the Web logs was explained to them, and they had already implemented a minor change to WSA in their previous assignment, the application domain was not what they were used to. Other courses focus on traditional areas, such as compilers and operating systems. Even the networking course focuses on network fundamentals, not necessarily net-centric applications. Consequently, effort estimation without any previous experience was handicapped from the start.

Although the application domain was new to the students, the essential problem remained. Unclear requirements lead to unsatisfactory effort estimates. In other words, requirements engineering for net-centric applications is just as important as in other application domains—perhaps even more so, given the novelty of the domain and the technology. This should not be a surprising conclusion, since requirements drive so much of any software project.
Were the requirements stated fairly? Perhaps not, but they were realistic. I deliberately

did not state the requirements in a formal specification, in part because the students had not yet learned about such notations. The requirements were stated in typically ambiguous prose. Consequently, most students did not really understand what I meant, and this was reflected in their erratic effort estimates.

About the author

Scott Tilley is a Visiting Scientist with the Software Engineering Institute at Carnegie Mellon University, an Assistant Professor in the Department of Computer Science at the University of California, Riverside, and Principal of S.R. Tilley & Associates, an information technology consulting boutique. He can be reached at stilley@sei.cmu.edu. The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

Bugs or Defects?

NEWS AT SEI

Author

Watts S. Humphrey

This library item is related to the following area(s) of work:

Process Improvement

This article was originally published in News at SEI on: April 1, 1999

One of the things that really bothers me is the common software practice of referring to software defects by the term "bugs." In my view, they should be called "defects." Any program with one or more defects should be recognized as defective. When I say this, most software engineers roll their eyes and feel I am out of my mind. But stick with me and you will see that I am not being unrealistic.

To explain why the term "bug" bothers me, I need to pose three questions. First, do defects really matter? Second, why not just worry about the important defects? And third, even if we have to worry about all the defects, why worry about what we call them?

Do Defects Really Matter?

To answer this question, we first need to find out if defects are or can be serious. To do that, we must define what we mean by "serious." Here, while there will be wide differences of opinion, there is almost certainly a basis for general agreement. First, if a defect kills or seriously injures someone, we would all likely agree that it was a serious defect. Further, if a software defect caused severe financial disruption, we would all probably agree that it too was a serious defect.

Lastly, there are lots of less significant problems, such as inconvenience, inefficiency, and just plain annoyance. Here, the question of seriousness obviously depends on your point of view. If you are the one being inconvenienced, and if it happens often, you would likely consider this serious. On the other hand, if you are the supplier of such software, and the inconveniences do not cause people to sue you or go to your competitors, you would probably not judge these to be serious defects. However, if

Related Links

News

SEI to Co-Sponsor 26th Annual IEEE Software Technology Conference

SEI Cosponsors Agile for Government Summit

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

these defects significantly affected the bottom line of your business, you would again agree that these too were serious defects.

So we are left with the conclusion that if defects cause loss of life or limb, result in major economic disruption to our customers or users, or affect the profitability of our businesses, these are serious defects.

Do They Matter To You?

The real question, however, is not whether defects matter in a theoretical sense but whether they matter to you. One way to think about this would be in terms of paying the costs of dealing with defects. Suppose you had to pay the discovery, recovery, reporting, repairing, redistribution, and reinstallation costs for every defect a customer reported for your programs. At IBM in my day, these costs averaged around $20,000 per valid unique defect. And there were thousands of these defects every year.

Of course, the problem is not who made the mistake but why it wasn't caught by the process. Thus, instead of tracing customer-discovered defects back to the engineers who injected them, we should concentrate on fixing the process. Regardless of their causes, however, defect costs can be substantial, and if you had to personally pay these costs, you would almost certainly take defects pretty seriously.

Why Not Just Worry About the Serious Defects?

At this point, I think most people would agree that there are serious defects, and in fact that the reports of serious defects have been increasing. Now that we agree that some defects are serious, why not just worry about the few that are really serious? This question leads to a further question: Is there any way to differentiate between the serious defects and all the others? If we could do this, of course, we could just concentrate on the serious problems and continue to handle all the others as we do at present.

To identify the serious defects, however, we must differentiate them from all the others. Unfortunately, there is no evidence that this is possible. In my experience, some of the most trivial-seeming defects can have the most dramatic consequences. In one example, an executive of a major manufacturer told me that the three most expensive defects his organization had encountered were an omitted line of code, two characters interchanged in a name, and an incorrect initialization. These each caused literally millions of dollars of damage. In my work, some of my most troublesome defects were caused by trivial typing mistakes.

How Many Defects Are Serious?

Surprisingly, the seriousness of a defect does not relate to the seriousness of the mistake that caused it. While major design mistakes can have serious consequences, so can minor coding errors. So it appears that some percentage of all the defects we inject will likely have serious consequences. But just how many defects is this? And do we really need to worry that much about this presumably small number?

Based on my work with the Personal Software Process (PSP), experienced programmers inject one defect in about every 10 lines of code (LOC) [Humphrey 95]. While these numbers vary widely from engineer to engineer, and they include all the defects, even those found in desk checking or by the compiler, there are still lots of defects. And even for a given engineer, the numbers of defects will vary substantially from one program to the next. For example, when I have not written any programs for just a few weeks, I find that my error rate is substantially higher than it was when I was writing pretty much full time. I suspect this is true of other programmers as well. So engineers inject a large number of defects in their programs, even when they are very experienced.

Won't the Compiler Find Them?

Now, even though there are lots of defects, engineers generally feel that the compiler will find all the trivial ones and that they just need to worry about the design mistakes. This, unfortunately, is not the case. Many of my programming mistakes were actually typing errors. Unfortunately, some of these mistakes produced syntax-like defects that were not flagged by the compiler. This was because some of my mistakes resulted in

syntactically correct programs. For example, typing a ")" instead of a "}" in Pascal could extend a comment over a code fragment. Similarly, in C, typing "=" instead of "==" can cause an assignment instead of a comparison. From my personal data, I found that in Pascal, 8.6% of my syntax defects were really syntax like, and in C++, this percentage was 9.4%.

Some syntax-like defects can have serious consequences and be hard to find. If, as in my case, there are about 50 syntax defects per 1000 lines of code (KLOC), and if about 10% of them are actually syntax like, then about 5 or so defects per KLOC of this type will not be detected during compilation. In addition there are 30 to 50 or so design-type defects that will not be detected during compilation. So we are left with a large number of defects, some of which are logical and some of which are entirely random. And these defects are sprinkled throughout our programs.

How About Exhaustive Testing?

Next, most engineers seem to think that testing will find their defects. First, it is an unfortunate fact that programs will run even when they have defects. In fact, they can have a lot of defects and still pass a lot of tests. To find even a large percentage of the defects in a program, we would have to test almost all the logical paths and conditions. And to find all of the defects in even small programs, we would have to run an exhaustive test.

To judge the practicality of doing this, I examined a small program of 59 LOC. I found that an exhaustive test would involve a total of 67 test cases, 368 path tests, and a total of 65,536 data values. And this was just for a 59 LOC program. This is obviously impractical. While this was just one small program, if you think that exhaustive testing is generally possible, I suggest you examine a few of your own small programs to see what an exhaustive test would involve. You will likely be surprised at what you find.

Just How Many Defects Are There?

So now, assuming you agree that exhaustive testing is impossible, and that some of these defects are likely to be serious, what next? First, we find that the programs that engineers produce have a lot of defects, at least before compilation and unit test. Also, we find that compilation and unit testing cannot find all of the defects. So a modest percentage of the defects are left after unit testing. But how many are likely left, and do these few defects really matter?

Here again, the data are compelling. From PSP data, we find that engineers typically find between 20 to 40 defects per KLOC when they first test their programs. From my personal experience, I also find that unit testing typically finds only about 45% to 50% of the defects in a program. This means that after unit test, programs will typically still have around 20 or more defects per KLOC remaining. This is an extraordinary number. This means that after the first unit testing, programs will typically have a defect every 50 or so LOC! And after integration and product test, there would still be 5 to 10 or more defects left per KLOC. Remember, all of these defects must be found in system testing, or they will be left for the customer.

Consider Some Data

Just so you know that this is not an exaggeration, consider some data from the Jet Propulsion Laboratory (JPL). This organization develops the spacecraft that NASA sends to explore the solar system. One of their internal reports listed all the defects found in the system testing of three spacecraft [Nikora 91]. These software systems all had about 20 KLOC, and they were each tested for two or more years. The cumulative defects per KLOC found during this testing by week are shown in the figure. As you can see, they all had from 6.5 to nearly 9 defects per KLOC found in system test, and that does not guarantee that all the defects were found. In fact, from the figure, it seems pretty obvious that some defects remained. Note that these data were taken after the programs were developed, compiled, unit tested, and integration tested.

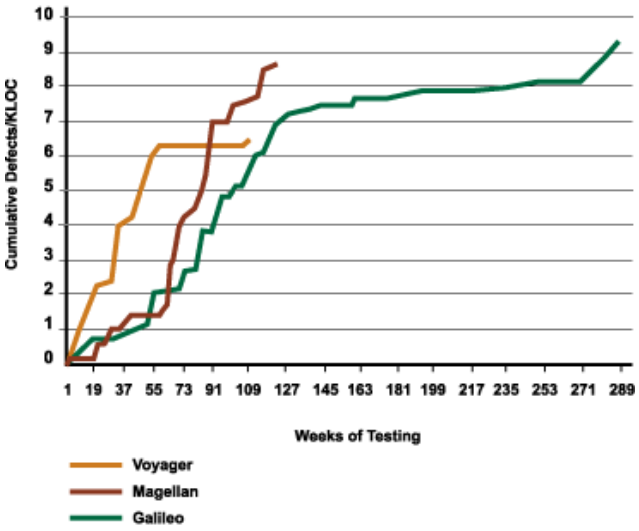

Figure 1: Spacecraft System Test Defects/KLOC

This, by the way, should not be taken as critical of JPL. Their programmers are highly skilled, but they followed traditional software practices. Normal practice is for programmers to first produce the code and then find and fix the defects while compiling and testing.

But Why Not Call Them "bugs"?

So, by now, you presumably agree that some defects are serious, and that there is no way to tell in advance which ones will be serious. Also, you will probably agree that there are quite a few of these defects, at least in programs that have been produced by traditional means. So now, does it matter what we call these defects? My contention is that we should stop using the term "bug." This has an unfortunate connotation of merely being an annoyance; something you could swat away or ignore with minor discomfort.

By calling them "bugs," you tend to minimize a potentially serious problem. When you have finished testing a program, for example, and say that it only has a few bugs left, people tend to think that is probably okay. Suppose, however, you used the term "land mines" to refer to these latent defects. Then, when you have finished testing the program, and say it only has a few land mines left, people might not be so relaxed. The point is that program defects are more like land mines than insects. While they may not all explode, and some of them might be duds, when you come across a few of them, they could be fatal, or at least very damaging.

We Think in Language

When the words we use imply that something is insignificant, we tend to treat it that way, at least if we don't know any better. For example, programmers view the mistakes they make as minor details that will be quickly found by the compiler or a few tests. If all software defects were minor, this would not be a problem. Then we could happily fix the ones we stumble across and leave the rest to be fixed when the users find them. But by now you presumably agree that some software defects are serious, that there is no way to know in advance which defects will be serious, and that there are growing numbers of these serious defects. I hope you also agree that we should stop referring to software defects as "bugs."

Finally, now that we agree, you might ask if there is anything we can do about all these defects? While that is a good question, I will address it in later columns. It should be no surprise, however, that the answer will involve the PSP and TSP (Team Software Process) [Webb 99].

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of Dan Burton, Bob Cannon, Sholom Cohen, Frank Gmeindl, and Bill Peterson.

In Closing, An Invitation to Readers

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. I am, however, most interested in addressing issues you feel are important. So please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them when I plan future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey

watts@sei.cmu.edu

References

[Humphrey 95] Humphrey, Watts S. A Discipline for Software Engineering. Reading, Ma.: Addison Wesley, 1995.

[Nikora 91] Nikora, Allen P. Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three JPL Flight Projects. Software Product Assurance Section, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, November 5, 1991.

[Webb 99] Webb, Dave & Humphrey, W.S. "Using the TSP on the TaskView Project," CROSSTALK, February, 1999.

About the Author

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

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

Doing Disciplined Work

NEWS AT SEI

Author

Watts S. Humphrey

This library item is related to the following area(s) of work:

Process Improvement

This article was originally published in News at SEI on: June 1, 1999

Over the years, I have often been asked about how to get management support for process improvement. Typically, engineers want to use better software methods; but they have found that either their management doesn't care about the methods they use or, worse yet, some managers even discourage them from trying to improve the way they work. In addressing this subject, I have decided to break it into two parts. The first part concerns disciplined work: what it is, and what it takes to do it. Then, in later columns, I will address the problems of getting management support for process improvement.

In discussing disciplined work, I answer five questions. First, what do we mean by disciplined work? Second, why is disciplined work important? Third, what are the elements of disciplined engineering work? Next, if you have the basic training and motivation to do disciplined work, why can't you just do it? And finally, what kind of support and assistance do you need to consistently do disciplined work?

What is Disciplined Work?

Discipline is an aspect of behavior. It involves consistently using sound methods. Discipline is defined as an activity or regimen that develops or improves skill; acting in accordance with known rules and proven guidelines. Disciplined behavior is generally needed whenever human error can cause harm, substantial inconvenience, or expense. The disciplined behaviors are then designed to reduce human errors, prevent common mistakes, and improve the consistency of the work. Also, many people are surprised to find that

Related Links

News

SEI to Co-Sponsor 26th Annual IEEE Software Technology Conference

SEI Cosponsors Agile for Government Summit

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

disciplined behavior generally improves efficiency, saves time, and even facilitates creativity.

(< 5 minute) survey.

Why is Disciplined Work Important?

In every advanced field, you get better and more consistent results by using proper methods and applying known and proven techniques. This is true in factories, development organizations, and even research laboratories. No one would agree to an operation by a doctor who had not finished medical school, spent years as a resident, and been board certified. Similarly, when hiring an accountant, you would not consider someone who did not have a CPA certificate and was properly licensed. If you had to go to court, you could not use a lawyer who had not passed the bar and been qualified to practice in the state. Anyone doing biomedical or nuclear research knows that disciplined behavior can be a matter of life and death.

While the software field is too new to have qualification mechanisms like those in many other fields, we now know the practices required for good software engineering. In addition, we also know that when software engineers use these methods, they consistently produce quality products on their committed schedules and for the planned costs. Unfortunately, however, these methods are not yet generally taught in university curricula. Thus, to advance in this field, engineers need to get their own training and to develop their personal skills.

What Are the Elements of Disciplined Engineering Work?

Disciplined software engineering involves more than just producing good technical results. As is true in other advanced fields, you need to address every aspect of the job. While technical competence is essential, you also need to consider the customer's needs, handle business concerns, and coordinate with your teammates. If, for example, you handle the technical concerns but ignore those related to the customer, you will likely produce a product that solves the wrong problem. While you might not lose your job, it is never good for an engineer's career to be associated with a failed project.

Proper attention to customer-related issues requires understanding the requirements before starting the design, maintaining close customer contact throughout the work, and planning and negotiating every change with the customer. Important business issues involve planning, tracking, and reporting on the work; focusing on quality from the beginning; and identifying and managing risks. Key teamwork issues relate to agreeing on goals, making and meeting commitments, and reviewing and supporting teammates' work. Finally, your team needs a logical development strategy, a sound architecture, a comprehensive design, and a set of rigorously followed standards and methods.

Getting the Needed Skills

There are various ways to obtain the skills needed for disciplined work. While some of these skills can be learned in university programs[1], many must be learned through on-the-job experience.[2] Instruction in disciplined personal and team software methods can also be obtained from the SEI and its transition partners, but that requires your management's support. While qualified training is the route that I would suggest for anyone who can follow it, a principal concern of many engineers is that they can't get management support.

Why Not Just Do It?

While getting training is important, consistently using the methods that you know is even more important. Unfortunately, it is also much more difficult. A growing number of engineers are being trained in disciplined engineering methods, but many of them find that even though they know how to do good software work, they are unable to practice what they know. The reason is that disciplined work is very hard to do, particularly when you try to do it by yourself. Here, there are two contributing elements: lack of personal discipline and inadequate coaching and support.

Personal Discipline

How many times have you decided to do something but never really did it? Like New Year's resolutions, there are lots of things we know we should do like quitting smoking, not eating between meals, exercising every day, and many others. We kid ourselves that we could do these things if we really had to, but somehow we never do.

The problem here is that most of us try to maintain strict personal disciplines all by ourselves. For example, can you visualize working for years to become a concert-quality pianist in a deaf world? When the quality of your work is invisible and nobody knows or cares how you perform, it is almost impossible to follow rigorous personal disciplines. This is why professional athletes and performing artists have coaches, trainers, conductors, or directors.

Even at the pinnacle of their fields, professional performers need the help and support of a cheering section, the constant push and motivation of a coach, and the demanding guidance of an informed and caring trainer. This is not just a nicety; it is absolutely essential. We humans are a group species. We work best in groups, and we have great difficulty performing alone. We need somebody who knows and cares.

Coaching and Support

I was fortunate to be on a marvelous team when I was in college. Our coach had been on the U.S. Olympic wrestling team; and he was an energetic, enthusiastic, and terribly demanding coach. Nobody wanted to disappoint him. We all worked harder than any of us had ever worked before. In our first year, he took a team of rookies to the AAU championship of 13 states. What was most interesting to me was that the next year I transferred to a different school. The wrestling coach was a nice guy, but he was not demanding or enthusiastic. Not only didn't the team do well, I didn't either. Superior coaching makes an extraordinary difference, and it is necessary for any kind of disciplined personal work.

What Kind of Support Do We Need?

The issues that we face in software engineering are severalfold. First, our field has not yet developed a tradition of disciplined work. Thus we must change an industry-wide culture. Second, coaching is not a common management style. Managers in software, as in other fields, feel more natural acting like straw bosses. Few know how to use, build, and develop the skills of their people. But this is the essence of management: helping and guiding people to do the best work that they are capable of producing. When people don't perform as well as they should, managers should help them develop their skills and motivate them to rigorously use the methods they know.

In software engineering, good work requires engineering discipline, and disciplined work requires coaching. In a subsequent column, I will discuss getting managers to act like coaches.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of Bob Cannon, Bill Peterson, and Mark Paulk. I also thank Jean-Marc Heneman for his questions and comments on this topic.

In Closing, An Invitation to Readers

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. I am, however, most interested in addressing topics that you feel are important. So please drop me a note with your comments, questions, or suggestions (watts@sei.cmu.edu). I will read your notes and consider them when I plan future columns. Thanks for your attention and please stay tuned.

[1]	Available PSP courses teach planning, process, measurement, and quality methods, using my text: *A Discipline for Software Engineering*. If you can't get into such a course, you could learn the basics from the PSP introductory text, *Introduction to the Personal Software Process.*
[2]	I have written a textbook for a new teamworking course. It is called *Introduction to the Team Software Process*, and it teaches the basics of the TSP. This book will be available from Addison-Wesley in late 1999.

About the Author

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

Contact Us
info@sei.cmu.edu
412-268-5800

Were You Ready for the Melissa Virus?

NEWS AT SEI

Author

Moira West Brown

This article was originally published in News at SEI on: June 1, 1999

The previous column in this series discussed the need for all organizations to be prepared to prevent and respond to computer security incidents. It pointed out that even having good security measures in place would not prevent your organization from suffering computer security incidents and explained the need for you to be proactive about detecting and responding to incidents. Soon after this column was published in the March 1999 issue of *news@sei*, a Microsoft Word macro virus named "Melissa" was released on the Internet.

In this issue of the column, I am joined by Katherine Fithen of the CERT® Coordination Center (CERT®/CC). We will discuss how well the Internet community was prepared to respond to the Melissa virus and how the Internet community can better prepare for similar or even more damaging events in the future.

The Melissa Word macro virus

The operation of the Melissa Word macro virus and the techniques to prevent it are well-documented. If you would like to learn more, visit our Web site and read our advisory at http://www.cert.org/advisories/CA-99-04-Melissa-Macro-Virus.html, and the frequently asked questions (FAQ) document at http://www.cert.org/tech_tips/Melissa_FAQ.html.

What was different about the Melissa virus?

Macro viruses first came on the scene in 1995, so it's reasonable to ask how the Melissa virus was so different from previous viruses that it merited

worldwide media attention and a hearing in the U.S. House of Representatives. The creator of the Melissa virus may or may not have realized its potential impact, but three unique aspects made it especially virulent:

- its speed of propagation
- the fact that it could be received from a known or trusted source
- its potential to disclose sensitive information

Speed of propagation

Unlike a worm, which is self-replicating, the Melissa virus like many viruses requires human action to propagate and cause new infections. However, previous macro viruses propagated by users who unknowingly triggered executable code usually resulted in the virus spreading to other files on the user's system or to other users *one at a time*. So, how could the Melissa virus spread so rapidly?

If triggered, the Melissa virus spreads rapidly by attempting to mail itself to the first 50 entries in every Microsoft Outlook MAPI address book available on the user's computer. The potential scaling of the virus was greater than 50 to 1 because some users have access to multiple address books. This means that if any of the first 50 entries in the address books were mailing lists, the virus would send itself to all the recipients of those mailing lists! This unique propagation strategy, coupled with the fact that Microsoft Word and Outlook are widely used by organizations on the Internet, meant that the Melissa virus had a vast base of agents already installed to support its propagation.

Many organizations have mailing list aliases named "all" that reach every staff member. One large commercial organization that uses Microsoft Outlook reported that its organization-wide mailing lists were the first listings in many of its staff members' address books. As a result the virus spread to every individual in the organization. Another site reported 32,000 copies of mail messages containing the Melissa virus spreading through its systems within 45 minutes!

Received from a known or trusted source

Many Internet users are wary about opening files or executing attachments from a person they do not know. This has resulted from a growing awareness among users of the threat and impact of viruses. The proliferation of spam and of unsolicited commercial email (i.e., the electronic equivalent of "junk mail") also has made users aware of potential threats lurking in email attachments.

Many of us use electronic address books to store frequently used email addresses. Because the Melissa virus infected attachments sent from the user's email account to recipients who likely knew and trusted the sender, the recipients were much less likely to suspect the message attachment contained a virus. The subject line and message text of an infected email message instructed the recipient to open the attached Microsoft Word document. Many users opened the document because they recognized the sender's name and believed that it was intentionally sent. However, neither party realized that the message contained the Melissa virus.

Potential to disclose sensitive information

The Melissa virus infects the Microsoft Word Normal.dot template. All Word documents use this template by default, so every document later created using this template will also become infected and be subsequently sent to the first 50 address book entries as described above. Needless to say, this hurt some

businesses by leaking documents containing sensitive information.

What was the extent of the damage?

The aspects described above gave the Melissa virus the capacity to spread rapidly and wreak havoc. But it was not as bad as it could have been, and it did not disrupt networked computer operations on the scale anticipated. Why not? Three factors helped to limit the spread of the Melissa virus: timing, effective response, and the role of the media.

Timing

In the United States, the virus was released on the afternoon of Friday, March 26, 1999. Because of the time zones, people in many other parts of the world had already left work for the weekend. In the U.S., many people leave work early on Fridays, and many universities in the U.S. were on spring break; so a large number of students and staff were on vacation. It was also the weekend before Easter, which starts a popular vacation period for many Europeans. Therefore, for a virus that requires human intervention to support its propagation, the timing significantly limited its effective spread.

Many organizations used this timing to their advantage to combat the virus. They asked their information technology (IT) staff to work through the weekend, disconnected their computers from the network, and began to clean their systems, update their virus scanners, and implement other preventive methods.

Effective response

Through coordination and cooperation, the international incident response community, the anti-virus vendors, and the media played major roles in limiting the spread and impact of the Melissa virus. The CERT/CC played a pivotal role in this response.

A brief timeline of events tells the story that unfolded and illustrates the contribution of the CERT/CC. The timeline covers the period from the Melissa virus release on Friday, March 26, through the response efforts over the following weekend.

The CERT/CC was alerted to the existence of the Melissa virus when it received its first incident report from an insurance company at 2:00 p.m. A second report came in at 3:30 p.m. from a member of FIRST (Forum of Incident Response and Security Teams; see http://www.first.org). By 5:00 p.m., the CERT/CC had received five reports and had started analyzing a copy of the virus. Another FIRST member shared a brief analysis of the virus around 6:00 p.m. and by 7:00 p.m., realizing the seriousness of the threat from the Melissa virus, the CERT/CC recalled staff to the office.

By midnight, CERT/CC staffers completed a thorough analysis of the Melissa virus and wrote the initial CERT advisory on the virus. The advisory was released at 5:30 a.m. on March 27. CERT/CC staff members were also working to reach many groups within the anti-virus community. They did this to ensure that the virus analysis was complete and that details from as many anti-virus vendors as possible could be included in the CERT advisory on the Melissa virus. The anti-virus community has a long history of collaboration and prompt response. This approach facilitates its response as it shares virus exploits and analyses to ensure that each vendor can implement its own identification and eradication techniques.

CERT advisories reach a vast number of system and network administrators on the Internet. But given the timing of the virus release and of the advisory, there

was concern that many people would be enjoying their weekend and would be oblivious to the growing threat from the Melissa virus. So the CERT/CC contacted the media to further alert the world to the potential impact of the virus and to recovery strategies.

The role of the media

Many incident response and security teams are wary of working with the media as part of their incident response strategy. Internet security topics greatly interest the media, but unless the quality of the reporting of these events is high, there is always a possibility that the technical information reported could be incorrect and that the coverage might do more harm than good.

In the case of the Melissa virus, though, the media reported the story accurately and effectively. This good reporting dramatically increased the world's awareness of the threat and the availability of appropriate countermeasures. The media coverage reached many system and network administrators responsible for IT system security, and many computer users began to understand the urgency of the situation as they learned about the virus over the weekend. Accurate news coverage provided users with the information that they needed to return to work and begin recovery and mitigation strategies. This greatly helped prevent further spread of the virus.

The effect of the widespread media coverage is evident from the CERT/CC's Web site statistics. Within the first 7 hours of the release of the Melissa advisory, it was downloaded 30,000 times from the CERT/CC Web site. After media coverage of the Melissa virus, the advisory was downloaded an additional 188,000 times the following Monday and 160,000 times Tuesday.

Can we respond more effectively next time?

As the Internet has grown, a smaller percentage of its users remember the Internet worm incident of 1988 that resulted in much of the Internet being completely unavailable. There was no infrastructure in place to respond to the worm, and it became the impetus behind the formation of the CERT/CC. (For more information, see ftp://coast.cs.purdue.edu/pub/doc/morris_worm/FAQ). Similarly, the Melissa virus incident serves to remind the Internet community of the potential for threats and the need to be better prepared to handle them.

A lucky combination of factors helped limit the Melissa virus's damage, and the overall response from the current response infrastructure was excellent. *However, the Internet is still vulnerable to far greater threats than the Melissa virus*.

Although the volume of email traffic that the Melissa virus generated resulted in a denial of service (i.e., an organization being unable to process email or access computer networks) for some, its impact could have been far worse. This would have been the case if the Melissa virus did as much damage as a virus like the CIH virus (see http://www.cert.org/incident_notes/IN-99-03.html). Viruses of this caliber completely destroy data and can render systems inoperable. A worse situation could also be caused by a virus that has a more stealthy approach or does not require a human action to propagate it.

If the Melissa virus had been released on a Monday rather than a Friday, it would have spread much more rapidly and the response community would have been under even greater pressure to respond quickly. The scale of the Melissa virus incident as reported to the CERT/CC was more than 300 organizations and 100,000 hosts. However, this was not as widespread an incident as it might have been. The CERT/CC has received reports about automated scans of Internet hosts where *tens of thousands* of systems are probed.

Although the Melissa virus posed a threat to many individual organizations and users, it posed far less of a threat to the overall Internet infrastructure. Unlike the Internet worm incident in 1988, the Internet was available as a communications medium to facilitate the response to the Melissa virus incident. If in the next big security incident the Internet is not available, the response community could still cooperate with the media to share information about the threat and the appropriate response (although even the media relies greatly on the Internet). However, if the Internet is not available because of a severe security event, it would be extremely difficult to analyze the technical cause of the problem, or to make countermeasures and patches available for downloading. Much of the analysis and coordination in response to the Melissa virus relied on the Internet infrastructure as a communications channel. This included the ability to share exploits, analysis, vendor communications, and advisories.

Sites used the Internet not only as a communications medium to pull down the CERT advisory but also to download updates for their virus scanners. If the Internet had not been available, access to those virus scanner updates would have been far more problematic.

Future viruses and other security events could occur on a much greater scale and cause more damage to the Internet community than the Melissa virus. We must continue to be aware of potential threats and continue to prepare for the potential disruption. Many organizations and response teams will evaluate their response to the Melissa virus and use the lessons learned to evaluate their readiness to respond to future threats. We encourage you to review your contingency and information technology security plans to ensure that you and your organization are better prepared for the next event.

About the authors

Moira J. West-Brown is a senior member of the technical staff within the CERT® Coordination Center, based at the SEI, where she leads a group responsible for facilitating and assisting the formation of new computer security incident response teams (CSIRTs) around the globe.

Before coming to the CERT®/CC in 1991, West-Brown had extensive experience in system administration, software development and user support/liaison, gained at a variety of companies ranging from academic institutions and industrial software consultancies to government-funded research programs. She is an active figure in the international CSIRT community and has developed a variety of tutorial and workshop materials focusing mainly on operational and collaborative CSIRT issues. She was elected to the Forum of Incident Response and Security Teams Steering Committee in 1995 and is currently the Steering Committee Chair. She holds a first-class bachelor's degree in computational science from the University of Hull, UK.

Katherine Fithenis the manager for the CERT Coordination Center. She has been a member of the CERT/CC since March 1992. The CERT Coordination Center provides technical assistance to Internet sites that have computer security issues or concerns, or that have experienced a computer security compromise. Prior to joining the CERT/CC, she was a user consultant for PREPnet, the regional network service provider for the state of Pennsylvania. Fithen has earned a bachelor’s degree in retail management, a master’s degree in personnel management, and a master’s degree in information science.

Share This Page

For more information

Contact Us

info@sei.cmu.edu

412-268-5800

Software Strategy for Technology Collaborations

NEWS AT SEI

Author

Mike Mattison

This library item is related to the following area(s) of work:

Process Improvement

This article was originally published in News at SEI on: June 1, 1999

Most companies today are involved in various forms of partnerships, alliances, and collaborations. "Technology collaborations" are formed specifically to improve the software development organization, in whatever capacity, whether it involves technical methods, processes, or practices. In all cases, these collaborations are difficult to manage, control, and track to productive closure. After more than 15 years of working with executive management teams, I've identified a pattern of factors that are essential to managing and guiding the technology collaboration, and particularly the formation of software strategy to improve the software engineering enterprise.

In this column, the first of two columns on this subject, I will consider why technology collaborations are essential to sustaining competitive advantages in software engineering development, and why technology transition and collaboration planning require the support of good strategy formulation and

Related Links

News

SEI to Co-Sponsor 26th Annual IEEE Software Technology Conference

SEI Cosponsors Agile for Government Summit

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

planning. In my column in the next issue of SEI Interactive, I will explore in detail the eight key factors that I consider essential to managing and guiding the technology collaboration activity.

Software drives value

Who would dispute the astonishing value of software to the competitive strategy of the firm? Whether your firm produces aircraft, phones, Web-based systems, satellites, or semiconductors, software is what drives a major portion of your "value proposition" in the market. Software impacts all corners of the business operation, from manufacturing and design to distribution and customer service. Software is what makes you unique. Software has the power to differentiate you in the market, to create a powerful and compelling identity among alternative products or services. The performance of software creates superior functions, drives feature performance, and achieves value. In many cases, software creates the product and interface value that your customers want, need, and will return to in the future. Whether your customers are looking for software that is fast, accurate, reliable, safe, or effective, your software capability has everything to do with delivering the lasting value proposition to your customers.

Software is so important that every company should have an evolving and continuous software strategy, a "master plan" that replaces management's old Ouija board and voodoo dolls for making decisions about software engineering. Yet, in my 15-plus years of experience working with corporate management teams, I have found that most organizations do not have a software strategy. In fact, few software groups have created a basic plan of action that outlines vital issues, such as team roles and responsibilities, training activities, improvement planning, management issues, or key performance goals.

Without a software strategy that encompasses the key functional areas of software performance, we cannot conduct meaningful evaluations or decision analyses regarding software technology collaborations. The software strategies are the "blueprints" that guide our technology collaborations, and keep our decisions on track to achieve business value from our software performance.

I'd like to offer an approach that focuses on *strategy development* as a basis for improving how we collaborate to achieve software innovations. The value of software

innovation is not bankable until that innovation succeeds in application. To be of value, it

has to work in the real world, not just in the lab. Also, such innovation must be enhanced

and accelerated, typically by technology collaborations with partner organizations with

credible expertise and the supporting resources that are necessary for investigating and

evaluating new software engineering technologies. Further, successful technology

collaborations for software do not happen by luck and fortune alone; those successes are

the result of careful development of software strategy and planning. Lastly, I am

presenting eight key factors that are essential to developing an effective strategy in

support of software technology collaborations.

The importance of technology collaborations

My primary goal at the SEI is to identify and develop corporate partners for technology

collaborations that advance the formation, introduction, and adoption of new technologies

for software engineering. This requires careful team development of our internal

technologists from various SEI functions, in concert with the executives, managers, and

software practitioners of candidate partner companies. Each collaboration must have

consensus and alignment of management, business impact, financial resources, staff

capacity, and an appropriate technical domain. The transition of new technologies into

technical practice is not a trivial matter, and requires the acceptance and support of

numerous vital players from each collaboration partner. Companies do not conduct

technical collaborations, or joint ventures, without careful analysis of the key issues,

including

· review of the business plan, financials, schedule, operational impact, and resources

· technical domain, project selection, staff credentials, and project control/tracking

· acceptance and sponsorship by management, and sustained executive endorsement

· expected business value and impact upon final system performance for the user or

customer

· legal/contractual issues, performance criteria, intellectual property, and liability

· new technology insertion and organizational, cultural, and change management

dynamics

Clearly, forming collaborations between organizations is a substantial business venture

that merits qualified business and technology management and planning. These

collaborations-with potentially enormous payoffs-do not happen based on a few phone calls and some technical exchange meetings. The value of each collaboration is based on key stages: analyzing critical success factors, developing the software strategy, and forming the detailed plans for each collaboration.

I have often invested 6 to 10 months working with our internal technical teams and our client organizations to formulate tailored work agreements by which to conduct each technology collaboration. Getting agreement on introducing new technology into software operations is a long, detailed, conflicting, and arduous task. Companies don't like to do anything to their software unless and until everyone in each department understands what will be done, to whom, and why. Otherwise, it simply makes a lot of people very nervous, and failure is inevitable. Like good civil engineering, successful construction execution is always a function of the quality and detail of the planning. In short, great blueprints are necessary to produce great architecture. That's the role of technology collaborations.

From lab to market

Great technology alone rarely makes the leap from lab to market unless strong business and operational management and planning guides the process. Innovation may happen by itself in a contained lab, but the value of that innovation is zero until the transition to practice is made complete by successful application. Good ideas need commercial partners to help demonstrate the level of value and viability, and to further enhance, evolve, and increase the target efficacy of the proposed new technology. It is the technology collaboration that creates a defined and structured process between organizations that seek to transition new technology from concept to practice. In every instance where a technical practice or process has been improved with a collaboration partner, there has been a well-managed and directed collaboration strategy and plan to support the difficult transition period.

For the SEI's purposes of improving the practice of software engineering, the core function of each collaboration is the transition of new technology from one collaborator to the other. The purposes may include proof-of-concept analyses for embryonic-level technical practices, or even basic R&D with a specific technical or industry domain. Another purpose may be to conduct a trial run or to pilot early-stage technology with

multiple collaborators to gather data and generate a trend analysis. Yet another purpose of

technology transition may be to introduce a viable commercial technology into a new

industry to further demonstrate the technology's robustness across new product and

application domains. The vital challenge of each collaboration is to successfully guide

and manage all aspects of the business and technical relationship of both parties.

Collaborations between or among organizations to improve the software engineering

function require a strategy. Collaborating without a clearly developed software strategy is

like building a house without a set of blueprints: it is a deception of activity without

meaningful results. Strategy is the blueprint to keep you on course to ensure that each

collaboration will yield the desired results of maximum technical and business value. To

simply say you're in collaboration is insufficient for achieving technology transfer

between organizations. Many organizations have tried this approach, and many have

failed. In considering the development, introduction, and adoption of new software

technologies, we expect a long-term effort (greater than one year), the participation of

numerous subject experts, a review by senior management, and a minimum investment of

typically $50,000.1

These collaborations are, after all, intended to explore the vague uncertainties of new

emerging technical competencies that are-at this early stage-unproven and unknown,

and that lack reliable economic analysis. The prize of first entry to market, however, is

potentially enormous for technical innovations in software engineering. Clearly,

collaborations at this level and scope of effort require a well-defined strategy to protect

and guide the core goals and principles of the effort over the long run. In my experience,

the best results stem from those organizations that develop an internal software strategy,

which begins by answering the following questions:

1 In my experience, the collaboration of 2 or more organizations with intent to form, introduce, and

adopt new technologies for software engineering requires a 3-month negotiation/agreement phase and

a minimum 6- to 12-month implementation phase, often followed by a 1-month post-mortem phase.

Level of effort requires participation by key executives, senior and middle managers, project leaders,

contract staff, legal, accounting, and software engineering subject experts and technical staff. Case

evidence varies widely, and we do not publish proprietary contract information, but collaborations of

this scope and level of effort routinely require, at a minimum, a $50,000 budget allocation to ensure

reasonable implementation and success of initial goals.

· Why are you conducting this collaboration?

· What aspect of your software do you expect to impact or improve?

· What resources and sponsorship have been negotiated to support this effort?

· What is the expected "business value" to be realized from this investment in new

technology?

· What is the expected impact of not conducting this collaboration?

Creating a software strategy is a challenge for software organizations. In working with

numerous corporate firms, I have rarely found such items as business plans, improvement

programs, training routines, and technical effort analyses combined into a software

strategy. Yet, it is a very useful and productive endeavor for any software organization,

and it is a requirement for guiding and sustaining a technical collaboration. Strategy

formulation is also the task of the software group, not consultants or corporate

management. Those who own the implementation role should develop strategy.

Strategy provides the guiding principles necessary to calculate your core business and

critical resources for achieving business goals among competitors in complex markets.

Strategy is vital to performing your operations in a sustained environment of change and

uncertainty; to ensuring continuous improvement while operating at peak production; to

balancing those attributes that are essential to your ability to achieve your core business

goals. Strategy may include many considerations, including finance, technology,

management roles, design, manufacturing, customers, distribution, pricing-those

attributes that impact your ability to achieve your core business goals. Above all, strategy

requires all interested parties to come to the negotiating table and achieve consensus and

alignment.

We may never agree on which considerations are exactly necessary to develop a software

strategy. The fact is that companies and their markets have many variant business drivers

and operating conditions, and a "universal strategy" cannot be expected to address software engineering issues across all conditions. But which attributes, or key factors, are vital to managing the technology collaboration process? How can we better understand which factors are the vital core to building an effective software strategy? While every company's software strategy should address those factors unique to its own business, software, and technical requirements, I've identified a pattern of factors that have repeatedly proven absolutely essential to software strategies that drive successful technology collaborations. These factors, and their goals, are listed below.

Factor	**Goal**
Technical issues	Define and understand the primary software technical issues of your organization.
Primary business drivers	Understand the key factors that drive your business and industry.
Improvement history	Understand which improvement programs have succeeded in the past, which have failed, and why.
Performance goals	Understand how your organization defines and manages performance goals for software engineering.
Executive owner	Understand the role of senior executive management in new software technology collaborations.
Management team	Understand how management is structured in relation to your firm's software development operations.
Software engineering	Understand your firm's software engineering capabilities, technical processes and methods, and overall engineering conditions, resources, platforms, and operations.
Customers	Understand how the software organization delivers value to the customer.

By using these key factors as a baseline analysis, organizations can formulate effective strategy in support of complex technology collaborations. In the next issue of *SEI Interactive*, I will examine each of these key factors in detail.

SEI Interactive, 06/99 page 7

http://interactive.sei.cmu.edu/Columns/Business/1999/June/Business.jun99.htm

About the author

Mike Mattison is a senior member of the Technical Staff at the Software Engineering Institute (SEI). Mike is currently managing a portfolio of technology collaborations with Fortune 500 client partners. The goal of his work is to accelerate the technical development and commercial transition of new software capabilities and methods into applied practice. He has directed a technology partnering strategy for the SEI that has produced successful software collaborations with Fortune 500 corporations in the aerospace, automotive, banking, telecommunications, defense, and software industries. Mike is presently researching critical software performance in extreme high-reliability domains. His area of interest is the impact of software management performance on sustainable competitive advantage in commercial industries. You can reach Mike at mvm@sei.cmu.edu.

Contact Us
info@sei.cmu.edu
412-268-5800

Using Scenarios in Architecture Evaluations

NEWS AT SEI

Author

Rick Kazman

This library item is related to the following area(s) of work:

Software Architecture

This article was originally published in News at SEI on: June 1, 1999

When we analyze software architectures, we always want to do so with respect to an explicit or assumed set of quality attributes: modifiability, reusability, performance, and so forth. Most software quality attributes are, however, too complex and amorphous to be evaluated on a simple scale, in spite of our persistence in describing them that way.

Consider the following:

- Suppose that a system can accommodate a new computing platform merely by being re-compiled, but that same system requires a manual change to dozens of programs in order to accommodate a new data storage layout. Do we say that this system is or is not modifiable?
- Suppose that the user interface to a system is carefully thought out so that a novice user can exercise the system with a minimum of training, but the experienced user finds it so tedious as to be inhibiting. Do we say that this system is usable or not?

Related Links

News

SATURN Conference Announces Additional Keynote, Conference Scholarships

Distinguished Speakers, Strong Technical Program Set for SATURN 2014

See more related news »

Training

Big Data - Architectures and Technologies

Documenting Software Architectures - eLearning

See more related courses »

The point, of course, is that quality attributes do not exist in isolation, but rather only have meaning within a context. A system is modifiable (or not) with respect to certain classes of changes, secure (or not) with respect to specific threats, usable (or not) with respect to specific user classes, efficient (or not) with respect to specific resources, and so forth. Statements of the form “This system is highly maintainable” are, in my opinion, without operational meaning.

This notion of context-based evaluation of quality attributes has led us to adopt scenarios as the descriptive means of specifying and evaluating quality attributes. For example, the Software Architecture Analysis Method [1] is a scenario-based method for evaluating architectures; it provides a means to characterize how well a particular architectural design responds to the demands placed on it by a particular set of scenarios, where a scenario is a specified sequence of steps involving the use or modification of the system.

It is thus easy to imagine a set of scenarios that would test what we normally call modifiability (by proposing a set of specific changes to be made to the system), security (by proposing a specific set of threat actions), performance (by proposing a specific set of usage profiles), etc.

A particular scenario actually serves as a representative for many different scenarios. For example, the scenario “change the background color on all windows to blue” is essentially equivalent to the scenario “change the window border decorations on all windows.” We use the clustering of scenarios as one of our evaluation criteria. As a consequence, judgment needs to be exercised as to whether the clustered scenarios

represent variations on a similar theme or whether they are substantially different. In the first case, the clustering is a good thing; in the second, it is bad. In other words, if a group of scenarios are determined to be similar, and they all affect the same component or components in an architecture (i.e., they cluster), we deem that to be a good thing, because it means that the system’s functionality has been modularized in a way that properly reflects the modification tasks. If, on the other hand, a group of similar scenarios affect many different components throughout an architecture, we deem that to be bad.

As an aid to creating and organizing scenarios, we appeal to the concept of roles related

to the system. Examples of roles include: the person responsible for upgrading the

software—the end user; the person responsible for managing the data repositories used by

the system—the system administrator; the person responsible for modifying the runtime

functions of the system—the developer; the person responsible for approving new

requirements for the system, etc. The concept of roles matches the difference between

runtime qualities and non-runtime qualities—that is, those qualities that are a function of

the system's execution (such as performance), and those that reflect operations performed

offline in a development environment (such as modifiability). Scenarios of the former

variety would be performed by roles such as the end user; the latter by developers or

maintainers.

In a perfect world, the quality requirements for a system would be completely and

unambiguously specified in a requirements document that is completed before the

architecture begins. In reality requirements documents are not written, or are written

poorly, or are not finished when it is time to begin the architecture. So the first job of an

architecture evaluation is to elicit the specific quality goals against which the architecture

will be judged.

If all of these goals are specifically, unambiguously articulated, that's wonderful.

Otherwise, we ask the stakeholders to help us write them down. The mechanism we use

is the scenario. A scenario is a short statement describing an interaction of one of the

stakeholders with the system. A user would describe using the system to perform some

task. A maintenance stakeholder would describe making a change to the system, such as

upgrading the operating system in a particular way or adding a specific new function. A

developer's scenario might talk about using the architecture to build the system or predict

its performance. A customer's scenario might describe the architecture reused for a

second product in a product line, or might assert that the system is buildable given certain

resources.

Scenarios guide elicitation

In addition to clarifying requirements, scenarios help to prioritize what parts of the architecture should be elicited first. The brainstormed list of scenarios is prioritized based on several criteria. Prioritization criteria might include the most important uses of the system, the most illuminating uses in terms of how much of the architecture is covered by the scenario, the most important attribute goals, and the attribute goals that are most difficult to achieve.

Scenarios realized as sequences of responsibilities

A scenario can be thought of as a structure that operationalizes the other structures. A scenario starts with a stimulus of some kind and then shows how the architecture responds to that stimulus by identifying and locating the sequence of actions or responsibilities for carrying out a response to the stimulus. Scenarios tie together functional requirements, the quality attributes associated with the functional requirements, and the various architectural structures, as were presented in my December 1998 column, "Representing Software Architecture"[3].

The process of mapping or overlaying scenarios onto the architecture's structures is an important aspect of many types of analysis. Modifiability analysis maps change scenarios onto the other structures as part of trying to understand the transitive closure of the change and the interaction among various types of changes. Reliability analysis starts by considering various failure scenarios. Security analysis starts by considering various types of threat scenarios. Performance analysis starts by considering a scenario for which a timing requirement is central.

Eliciting scenarios in design and analysis

Although we have been discussing scenarios that are specific to a single quality attribute, even quality-specific scenarios have an impact on multiple qualities. For example, consider the scenario: "change the system to add a cache to the client." It is not only legitimate, it is mandatory, to ask about the effect of this change on performance, security, or reliability. So one quality may be used to motivate creation of a scenario but then the impact of that scenario on other qualities must be considered.

Furthermore, the requirements come from many stakeholders. Why is this? No single stakeholder represents all the ways in which a system will be used. No single

stakeholder

will understand the future pressures that a system will have to withstand. Each of these concerns must be reflected by the scenarios that we collect.

We now have a complex problem however. We have multiple stakeholders, each of whom might have multiple scenarios of concern to them. They would rightly like to be reassured that the architecture satisfies all of these scenarios in an acceptable fashion. And some of these scenarios will have implications for multiple system qualities, such as

maintainability, performance, security, modifiability, and availability.

We need to reflect these scenarios in the architectural structures that we document and the architectures that we build. We need to be able to understand the impacts of the scenarios on the software architecture. We further need to trace the connections from a scenario to other scenarios, to the analytic models of the architecture that we construct, and to the architecture itself. As a consequence, understanding the architecture's satisfaction of the scenario depends on having a framework that helps us to ask the right

questions of the architecture.

To use scenarios appropriately, and to ensure complete coverage of their implications, we

typically consider three orthogonal dimensions, as shown in Figure 1.

Figure 1: A Scenario Elicitation Matrix

The entries in the matrix are the specific scenarios. This characterization allows us to manage the scenarios not only for specification of requirements but also for subsequent validation of the architecture that is designed. The initial use of a quality-specific scenario might be considered during the design step, but the impact of that quality scenario on other qualities is also important during the analysis step.

Scenarios guide analysis

In the Architecture Tradeoff Analysis Method [2], we use three types of scenarios to guide and inform the analysis: use cases (these involve typical uses of the existing system

and are used for information elicitation); growth scenarios (these cover anticipated changes to the system), and exploratory scenarios (these cover extreme changes that are

expected to “stress” the system). These different types of scenarios are used to probe a

system from different angles, optimizing the chances of surfacing decisions at risk.

Examples of each type of scenario follow.

Use cases:

1. *The ship's commander* requires his authorization before releasing certain kinds of weapons.

2. *A domain expert* wants to determine how the software will react to a radical course adjustment during weapon release (e.g., loft) in terms of meeting latency requirements?

3. *A user* wants to examine budgetary and actual data under different fiscal years without re-entering project data.

4. A user wants to have the system notify a defined list of recipients by e-mail of the existence of several data-related exception conditions, and have the system display the offending conditions in red on data screens.

5. *A tester* wants to play back data over a particular time period (e.g., last 24 hours).

Notice that each of the above use cases expresses a specific stakeholder's desires. (Scenarios normally do not include the stakeholder as we did above.) In some cases a specific attribute is also called out. For example, in the second scenario, latency is called out as being important. In other cases, the such as the following, growth scenarios are used to illuminate portions of the architecture that are relevant to the scenario:

1. Make the head-up display track several targets simultaneously.

2. Add a new message type to the system's repertoire.

3. Add collaborative planning: two planners at different sites collaborate on a plan.

4. Double the maximum number of tracks to be handled by the system.

5. Migrate to a new operating system, or a new release of the existing operating system.

The above growth scenarios represent typical anticipated future changes to the system. Each scenario also has attribute-related ramifications (other than for modifiability). For example, the following explanatory scenario will:

1. Add new feature 3-D maps.

2. Change the underlying platform to a Macintosh.

3. Reuse the software on a new generation of the aircraft.

4. Reduce the time budget for displaying changed track data by a factor of 10.

5. Improve the system's availability from 95% to 99.99%.

Each exploratory scenario stresses the system. Systems were not conceived to handle these modifications, but at some point in the future these might be realistic requirements for change.

Through the use of these scenarios, the software architecture is explored and analyzed. Use cases help in eliciting the architecture, understanding it, and analyzing its runtime qualities such as availability, security, and performance. Growth and exploratory scenarios help in understanding how the architecture will respond to future pressures, and thus aid in understanding the system's modifiability. Each is a crucial addition to gaining intellectual and managerial control over the important asset that a software architecture represents.

References

[1] R. Kazman, G. Abowd, L. Bass, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures," Proceedings of the 16th International Conference on Software Engineering, (Sorrento, Italy), May 1994, 81-90.

[2] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, "Experience with Performing Architecture Tradeoff Analysis", Proceedings of ICSE 21, (Los Angeles, CA), May 1999, to appear.

[3] R. Kazman, "Representing Software Architecture", *SEI Interactive*, Volume 1, Issue 3, December 1998.

About the Author

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

The views expressed in this article are the author's only and do not represent directly or imply any official

position or view of the Software Engineering Institute or Carnegie Mellon University. This article is

intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored

by the U.S. Department of Defense and operated by Carnegie Mellon University.

SM IDEAL, Interim Profile, Personal Software Process, PSP, SCE, Team Software Process, and TSP

are service marks of Carnegie Mellon University.

® Capability Maturity Model, Capability Maturity Modeling, CERT Coordination Center, CERT,

and CMM are registered in the U.S. Patent and Trademark Office.

Introduction: Checking In on a Process Improvement Revolution

NEWS AT SEI

This library item is related to the following area(s) of work:

Process Improvement

This article was originally published in News at SEI on: June 1, 1999

"A disciplined software engineering process includes both effective defect management and comprehensive planning, tracking, and analysis methods," wrote Watts Humphrey in

1995, in the preface to his book *A Discipline for Software Engineering*. "This book introduces these disciplines and shows you how to use them to do better development work on both small and large programs."

With that understated passage, Humphrey and the Software Engineering Institute launched the Personal Software Process℠ (PSP℠), a revolutionary new way for individual software engineers—and ultimately, software organizations—to produce computer programs. Humphrey and the SEI followed up with the Team Software Process℠ (TSP℠) in experimental form in 1996, recognizing that the individualized methods of the PSP often worked best when applied by entire teams.

In this issue

The Features section of this issue of *SEI Interactive* checks in on this revolution-inprogress. Our Background article, "CMM, PSP, and TSP: Three Dimensions of Software Improvement," offers a condensed version of Humphrey's three-part series in the

Related Links

News

SEI to Co-Sponsor 26th Annual IEEE Software Technology Conference

SEI Cosponsors Agile for Government Summit

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

February, March, and April 1998 issues of *CrossTalk* magazine (available online at

http://stsc.hill.af.mil/crosstalk). The article provides an overview of how PSP and TSP

can give individuals and teams the tools they need to achieve software process

improvement.

Our Spotlight article, “Delivering on the Promise of Process Improvement,” offers status

reports from successful PSP/TSP introductions at major organizations, including Boeing,

Motorola, and Hill Air Force Base. The Roundtable discussion, “Tales from the Front

Lines: Insights from PSP/TSP Trainers and Researchers,” provides a view of the

challenges of PSP/TSP introduction from the perspective of trainers and researchers who

know first-hand how hard it is to change engineers’ ingrained behaviors.

Finally, our Links feature offers a guided tour of information available on the Web about

PSP and TSP.

http://interactive.sei.cmu.edu/Features/1999/June/Introduction/Intro.jun99.htm

Roundtable Interview on PSP/TSP

NEWS AT SEI

This library item is related to the following area(s) of work:

Process Improvement

TSP

This article was originally published in News at SEI on: June 1, 1999

This article captures an exchange of ideas among people who conduct training in the Personal Software Process℠ (PSP℠) or Team Software Process℠ (TSP℠) for either academia or software development organizations, or who have conducted research related to PSP and TSP:

- Robert Cannon is a professor of computer science at the University of South Carolina and is currently a visiting scientist at the SEI, working in the Process Group.
- Gina Green is an assistant professor of information systems at Baylor University.
- William Hayes is a member of the technical staff at the SEI, working in the Software Engineering Measurement and Analysis (SEMA) group.
- Alan Hevner is an eminent scholar and professor in the Information Systems and Decision Sciences Department in the College of Business Administration at the University of South Florida.
- Thomas Hilburn is a professor in the computer science department at Embry-Riddle Aeronautical University, which offers an undergraduate degree in

Related Links

News

Heartbleed: Analysis, Thoughts, and Actions

TSP Symposium 2014 Goes Beyond Methodology to Focus on Software Quality

See more related news »

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

computer science and a master's in software engineering.

- Robert Pauwels is the commercial training director for Advanced Information Services.
- James Tomayko is director of the Master of Software Engineering program at Carnegie Mellon University.

Topics include

- experiences with PSP and TSP
- the reception of PSP and TSP; contributors to their success
- overview of a new SEI special report featuring PSP
- the importance of maintaining teams for training and later project work

- the challenge of diffusing PSP into the workplace
- the need for management persistence
- lessons learned from PSP and TSP training

Experiences with PSP and TSP

Bill Thomas (BT) (Moderator): The idea of this Roundtable is to have a group discussion about your experiences with training engineers to use the Personal Software Process℠ (PSP℠) and Team Software Process℠ (TSP℠), and to get an idea of how engineers and—for those of you in academia—students receive these new ideas. To start with, give us a brief description of your experience with PSP and TSP.

James Tomayko (JT): Watts Humphrey used my graduate students at Carnegie Mellon as guinea pigs for the first PSP course back in 1994. Their data appears in his textbook, *A Discipline for Software Engineering* [1995]. We've been using PSP since then and we require it prior to students arriving on campus. After they're admitted, I send them a package of CD-ROMs with lectures and other materials about PSP and TSPi. They take the course before they get here because we use TSP on all of our project teams in the software development studio. The studio is 40% of the curriculum for the year. We had one team last year and four teams this year.

Thomas Hilburn (TH): When Watts was first developing his original PSP book, he was on the Embry-Riddle advisory board and he convinced us to try out the draft that he was working on. One of my colleagues taught it. We thought it worked pretty well and

it's now part of the core—the first courses that entering master of software engineering students take. We've been using it for about four years.

A couple of years later, Watts was convinced that maybe we needed to introduce PSP at an earlier stage, so we also used an early draft that he had of the *Introduction to PSP*, sometimes called "PSP Lite," in our freshman sequence of courses. We've been using that for about three years with some modest success.

Last year, we started using TSP in the junior/senior-level course Introduction to Software Engineering, which is a broad-brush course with a project. For the first time this semester, we used it in our undergraduate senior design course.

I also worked at the SEI as a visiting scientist a couple of years ago in the process program and worked with Jim Over and Dan Burton, helping to develop some of the instructional material for the first PSP courses. I sat through the instructor's course. For

the last couple of years at Embry-Riddle, with the help of the SEI, we've developed and delivered a one-week summer workshop for faculty on PSP. Some of my colleagues have worked with Motorola and Boeing and several other organizations, helping them introduce PSP into their workplaces. I've had some involvement with that.

Robert Pauwels (RP): Advanced Information Services got involved with PSP in 1994 when we were looking for answers on how to apply the Capability Maturity Model® for Software (SW-CMM®) to small software projects or small software teams. We had been struggling with CMM. Watts Humphrey recommended that we take a look at PSP as a way of trying to progress and solve some of the issues. I was involved with the initial SEI train-the-trainer course in fall 1994. It's a required training course for our software engineers and project managers at our organization.

Currently, we have two sides to our business, the development group and the consulting group. As of yesterday, we have 100% of the people in our development group trained. Training in our consulting group is a little bit more spotty. We're trying to solve the issues in terms of how to get people to use PSP effectively when they're the only people who have been trained in it and when they're off working at a customer site and the customer has very chaotic processes, if anything at all.

At this stage, we have not utilized TSP, primarily because it was not around in 1994 when we were dealing with the issue of how to integrate PSP into our software development work. Somewhere down the road, we will give it a shot. It's a timing issue—we're trying to figure out which project to introduce it with.

Robert Cannon (RC): My first PSP training occurred in South Carolina, through the South Carolina Research Authority. I was trained at an industrial site, which is fairly rare because most PSP instructors have actually been trained in Pittsburgh. There was a special arrangement that allowed some instructors to be trained off site. Since then, I have taught the PSP class at my university. I intend to do that again in the fall and follow that with a Team Software Process course in the spring.

I had a tremendous opportunity this year to work with the PSP/TSP group. I've been involved in teaching virtually every one of the PSP and TSP offerings at the SEI. The opportunity to work at an industrial site has been extremely valuable for me. I'll be the coordinator for a summer faculty workshop, where we'll be doing essentially the same course that Tom did for the last two summers to train faculty to teach PSP. We're also going to have a separate workshop to help faculty plan to introduce TSP into their curricula, or at least to give a course based upon the new Team Software Process book by Watts Humphrey.

William Hayes (WH): I've been working with the PSP initiative as a member of the SEI's Software Engineering Measurement and Analysis (SEMA) Group to help collect data initially from the training courses and now from the field where engineers are using PSP on the job. I'm helping to characterize the impact, using some fairly rigorous statistical techniques, of what happens when you use the technology.

I've had occasion to visit with and interview a lot of folks in industry. I've also gone through the training myself and am currently applying some of the methods to my work.

Though I don't develop software, a lot of the discipline methods and data-collection techniques are actually quite applicable to a variety of settings. I'm using it in the impact study work that I'm doing now.

Alan Hevner (AH): I'm a professor at the University of South Florida. I started getting

interested in PSP because I taught a little bit of it in my classes at the graduate level for

master of science students. I didn't get into the full-blown course, but I did show it to the

students and it was quite interesting to them. Then when I started working with Gina

Green on her doctoral dissertation, we selected PSP as an innovative technology to study

our research model.

[*Editor's Note: The SEI special report "Perceived Control of Software Developers and*

Its Impact on the Successful Diffusion of Information Technology" by Alan R. Hevner

and Gina Green is based on the doctoral dissertation to which Hevner refers. The report

is available at: http://www.sei.cmu.edu/publications/documents/98.reports/

98sr013/98sr013abstract.html]

Gina Green (GG): As Al said, PSP was selected as the innovation that I wanted to

study in my doctoral dissertation, which looked at software development technique

diffusion.

The reception of PSP and TSP; contributors to their success

BT: From your perspective, how are PSP and TSP being received by software engineers?

For those of you working with students, how would you say they are being received by

students? Also, could you describe some of the circumstances that contribute to success

with those groups.

JT: I've been finding that by spending a lot more time at the very beginning, explaining

how we're actually teaching the process—we're not really teaching the programming

style—people are a little bit more oriented toward picking up the points. There is a lot of

process for relatively small problems during the training. Sometimes the students focus

on that instead of seeing the end result. We've had some problems giving appropriate tool

support for PSP. Our spreadsheets haven't been that good, but that's probably our fault.

On the other hand, using the TSP spreadsheet program that Jim Over developed—and

which is still being developed now—has worked very well for us. The detail I get from

that has been exceptional. Since we have a collaborative tool that we use on the Web to

keep track of all of our data and products within the various projects, the mentors at the

SEI and myself can see what's going on at any given time. We put those spreadsheets out

there so that we can see each cycle—we can see how things are going along. We can see

earned value, actual and predicted, the amount of time they're spending, how good their

estimates are. I can, on an individual basis, tell how somebody may have messed up in

underestimating or overestimating. Then we can talk about that because we have regular

meetings with individual students. So, we've found in cases where there are tools

available and where people understand that they're learning a process, it seems to be

pretty successful. If you leave them to their own, to invent their own tools or to spend too

much time concentrating on the problems themselves, that doesn't seem to work.

TH: We've worked quite a bit with Motorola. During training, the engineers accept it

very well. I'd say 90% of them think it's great and they're able to complete the PSP

material. The difficulty comes on the job: they're going into embedded systems, trying to

apply PSP or use it for maintenance. Adapting it to their job has been difficult. Some of

our faculty have worked to help them, mentor them, support them, and coach them.

That's helped, but that's the difficulty, actually bringing it into the workplace. This is a

place where management supports PSP. There were similar problems at Boeing. In the

classroom setting PSP, at the graduate level, goes over pretty well. Then we ask students

to use it in some other courses. It hasn't always been that effective. In the freshman-level

course, we struggle with how to motivate students and produce pretty good results. The

problem in our freshman courses is that students can't really see the improvements that

you get in a typical PSP course. The training effect has been pretty good, but the followup

in using it has been a little bit of a problem.

We started using TSP about two years ago and did two graduate student projects. Watts

came down and helped us launch development of a new scheduling program for our flight

line. We did a database program for the county assessor's office. They were both very

successful. They used early versions of the TSP and I think students really started to see

the value of PSP. I'd say those were successful and students were able to follow the

process and use the process to their benefit.

Software engineering is a junior/senior-level survey course. It usually has some sort of

team project involvement. I've been doing this for about eight years with marginal success. I used TSP last fall and it was the most successful as far as students learning to work as part of a team to develop a piece of software. In this case, it was a toy project,

using Watts's new text, *Introductory TSP*. I did two anonymous surveys. The students still didn't like filling out all the forms and collecting all the data, but they grudgingly admitted that it was worthwhile to do that. I did hear comments from a number of students, "Now I see why we study PSP." It has been, I think, a grand success so far.

RP: Our own organization reached the stage of institutionalization about two years ago. Not too many things are questioned anymore in terms of what we're trying to do with PSP and the engineers' acceptance of it. Everyone in the organization is doing it on the projects that we have management control over. Obviously we've got strong management ownership and support to make sure that it gets done. We've done quite a few things to integrate PSP into our culture to where, I think, the engineers really appreciate it, even though they still don't like filling out some of the forms. They really do appreciate how it helps them during the development work.

My experience has been in working with our own software development staff and with outside organizations. It's not uncommon at all—especially for experienced engineers—to be resistant, early on, to all the planning and all the forms you've got to fill out. But 90% of the time the data that's being collected and the feedback they get from that tends to turn the disbelievers into believers by the time the course is over. We recognize that there's a certain class of people who will never buy into these things no matter how much you try to convince them.

The theme of filling out forms seems to be a common, recurring issue. During PSP class, I have to spend time reintroducing most engineers to what a pen and pencil are. I had one engineer claim that he would get hand cramps from filling out the forms. Even though the course gets better and the support tools get better, there's still room for us to work on things and to make them even more acceptable. To a certain degree, a tool will ease that transition for people.

Other interesting reactions I get from students—especially as I start working with organizations that are much larger than our own—is that although many of them understand the value of PSP, they have total disbelief that their management will be able to successfully pull it off. They think that everything will go fine until the first schedule crunch comes around, and then managers will tell them to push all of this stuff aside and get back to it some day when they have more time on their hands. Organizations of that size are not far enough into implementation yet to find out whether or not that's actually going to be the case.

AH: In our survey, some people said they felt that in their training they received some reasonable tools and other people developed their own tools. We also got a number of comments like, "We can't be very productive with this unless the managers and development team agree on a set of tools that we'll all use." So, there is still some flux in

the field about what are the right tools and who should supply those tools.

RP: I think a certain class of people who put tools as the show-stopper may come from that 10% of the people who just don't want to do it. They're searching for excuses. On the other hand, I think we can appreciate that handy tools make it a lot easier for the amount of data that PSP wants us to collect.

RC: About half of my students would come and say, "I've really learned something about myself and about the way I tend to work." There were others who would never really want to use PSP unless they were in an environment where it was expected of them. I taught an industrial course, where unfortunately there was little management support, for programmers who didn't work in the traditional programming environment.

They were not all familiar with even C or Java or Pascal, for example. They kept insisting that because the problems were irrelevant to their work that they couldn't see why they should be doing it. The effort to motivate and say "We should all be working on common problems to collect data" was not one that was effective with them. If I ever worked in such an environment again, I'd want to be very careful to determine with management that they're much more supportive. In teaching an industrial course here at the SEI, I

found a very positive reaction to PSP by an extremely skilled group of programmers.

WH: I'm a research guy, so neat models that involve data easily catch my interest. It seemed logical that the software industry would want to take this up—why wouldn't they want predictability? But it's very difficult to really be honest with yourself and to instill that discipline to change the way you work. It means recognizing that you've got to keep your attention on it—you can't take a holiday from it. But to be that attentive to what you're doing requires a tremendous amount of concentration and discipline. I found it to be a very big challenge to use these methods. When I'm interviewing people in the field, I now have a different appreciation for what they're doing. I really enjoyed the course, though I struggled to complete it. It was *very* difficult for me to complete it.

I've seen people succeed very well with paper and pencil—they don't need a tool. I've also seen people with the greatest tools who don't collect the data because they don't want to be honest with themselves. I've seen people who were the only PSP-trained people in a chaotic group project and still hold on. I've seen groups of people working together who really benefit from having each other to talk to. At the same time, within organizations where there are groups that are successful, you'll find other engineers who struggle initially to try to keep the data, try to keep their investment in what they're doing. But when a management fire drill comes along, the message comes to them very clearly that spending time collecting data and getting better as a professional is not a high priority. Making this delivery is a higher priority. Senior management and middle-level managers can make that message very clear to engineers without having to be very

explicit about it. That's one of the things that really gets in the way. I think the comments we've heard thus far really support that as well.

TH: I think there's a little too much emphasis on the need for tools. I think tools are great for computations and rolling up the data. That's not the hard part. The hard part is that discipline—that relentless, continuous observation of the way you go about your work. Recording the time and defects just requires attention to details.

RP: Time tracking seems to be one that most organizations and engineers have a lot of reluctance to. They claim that they've got to have some sort of stopwatch tool in the

top

right-hand corner of their window or monitor or they just can't do it. It is perceived as too

much overhead to write down on a sheet of paper something like, "It's 1:40 and I'm

starting to code."

Overview of a new SEI special report featuring PSP

BT: Al and Gina, in your research on the diffusion of information technology you

surveyed software engineers about their experiences with PSP. Could you summarize that

project and your findings?

AH: We were interested in looking at some of the behavioral issues of what happens

when new technology gets put into place in organizations. Most people have looked at

end-users of technology. We wanted to look specifically at software development groups

and individuals and analyze a good innovative idea that they were using. We came on the

idea of using PSP because of its newness to the market and the idea that it was an

individual type of technology as opposed to a group or a team technology. Most of the

past research was on individuals, so this would allow us to relate what we found to some

past research in this area. The model that we developed is in the SEI special report.

GG: From a theoretical standpoint, one of our key findings is that a software developer's

sense of personal control in the software development process is very important. In fact,

we found that that was strongly related to the developer's overall satisfaction with using

PSP and the developer's use of PSP. PSP tended to provide a pretty high sense of

personal control to the software developer. That was good news from a theoretical

standpoint. From a practical standpoint, we chose PSP because there was a lot of interest

in it, but we found that the actual use of PSP was not as widespread as we had

anticipated.

AH: We wanted to find people who had used PSP on at least one actual development

project for at least three months and not more than two years. That made the use of PSP

an innovation. With that constraint, we eventually got good surveys from about 72 users.

GG: The survey respondents indicated that PSP did, in fact, improve the quality of the

software that they developed. In addition, they felt that PSP made them more conscious

of quality-related issues in software development. This means that PSP is fulfilling some

of its intended goals, at least as far as the respondents' perceptions go.

The importance of maintaining teams for training and later project work

BT: I want to ask a little bit about the issue of people being trained as a team and then

using PSP for projects together. What do you think is the importance of keeping team

members together both in the training and then in the project work?

JT: I think it's really critical. It's a way of getting positive peer pressure. I don't know if

"pressure" is the right word—maybe peer acceptance might be a better word. When we

introduce PSP, there are some who want to do it right away and some who don't want to

do it. There are some who sort of will do it if everyone else is doing it. Of course, we

have a clear advantage in that I can just order everybody to take the course. I could even

pay attention to whether they're using it later. But I still think that they've been much

more effective when they could use it as a group, when they could talk to each other in

the same language.

AH: We found in our survey that if developers felt that their training was individualbased,

it was difficult to implement because their work was team-based. Just looking

specifically at PSP, developers found that there was a disconnect between individual- and

team-based training.

TH: Watts Humphrey's new TSP book works very well concerning some of the

beginning mechanics of building an effective team. In industry or academia, these team

projects often don't have a process. They may have an organizational process, but not a

process that's attuned to teamwork. TSP guides you through some elementary things that

you'll want to do in the beginning, launch, planning, and strategy phases as a team.

Students can look ahead through a whole project and have some idea of what they're

going to be doing week by week. This provides some assurance and a basis for the team

to start understanding what they're going to do and communicating about that. Like so

much of PSP, it's really not that astonishing as far as the complexity. A lot of it is just

common sense and good guidance in a structured form. I think that's as important as

anything else in building a good team.

RP: Our experience is that if you do not train people in teams, PSP will not be used. Unless everyone on the team is trained—from the project manager through all of the engineers, and especially the project manager, being the leader of the team—it doesn't get used. That's generally a difficult and tricky thing to do because often organizations

are reluctant to shut down a project for a month or so as they get a team trained. They'd much rather steal one person from each of their teams and have them go through it. Then a few months later steal another person from each of their teams and have them go through it. Then, maybe, a year or so down the road, they can have everyone trained. That, in itself, leads to all kinds of additional problems. It's hard to pilot things like that unless everyone's ready to go at the same time. Our experience is that biting the bullet and doing it in teams is the most effective way.

RC: I'd like to also mention the Introduction to TSP. I think that students can spend a great deal of time figuring out what to do. To have their roles defined for them when they begin is extremely valuable. We really only have a short time—about 16 weeks in a semester—and that's not enough time to get anything of significance done even in the best circumstances. When you spend a few weeks just organizing and figuring out who is going to do what, you've lost valuable time. The fact that roles are defined so that there is minimal overlap helps a great deal. I believe that it would be true likewise for our TSP groups in industry.

JT: The organizational aspects have been really important. We have a longer time—we've got a year. The students do part-time in the fall and spring. They do about 12 hours a week. Then, they do full-time during the summer. Because of the pressure of the other core courses and electives and just getting organized, I was getting late deliverables. In many years, deliverables that were due in December were arriving in April on many projects. This year, I know that four teams used TSP for the first time and at the end of the second cycle, which would be somewhere in November, we had more things done related to the project that we normally would have had in April. That impressed me. All four teams were more productive than all of the teams prior to them, except for a couple

of really outstanding groups in the past that were just exceptions.

WH: One of the components of PSP training that's very powerful is the availability of data, not just your own data, but a way of judging your own data by looking at the group data. That team dynamic is an important component. Having said that, I have had occasion to interview engineers in the field who were very nervous about anyone seeing their data. I couldn't find any reason, through interviews with other people in that organization, that they should be nervous. Yet, they were nervous about sharing personal data with teammates for fear that they would misinterpret the information. I've also come upon engineers who have been using it for six months to a year and their first question to me, when they hand me their data is, "Is this any good?" They're not able to step back and evaluate it on an objective basis, or at least they don't feel confident in doing that. So, having peers to interact with, who don't threaten the protectiveness you might feel over your data, is a very important part of successfully adopting the PSP. TSP is one of the best mechanisms we can find to build in that peer interaction. You also have a coach whom you might view as a mentor, someone of some seniority, whose opinion you

respect. Having that person to help you analyze your data is a very important component to the continued use of PSP in industry.

TH: There is a lot of emphasis put on analyzing your data, but I think that's the most valuable part: the self-analysis. I know there's probably a fear that it's used to judge the effectiveness of individuals by management. I know that there are some groups in certain organizations that try to dispel that. At Motorola, even though management has said that they wouldn't use it as a means of evaluating individual performance, they did want to look at team PSP data. They decided that each team would elect a data security person who would collect the data, roll it up, and answer questions about individual data without associating it with individuals in a way that they could be identified. That is a fear and it's too bad because I think the data is very valuable for improving individual performance and, in the long run, team performance as we figure out ways to get team metrics. I'm not sure we have that yet.

The challenge of diffusing PSP into the workplace

BT: Gina and Al, the idea behind your research was to figure out why some of these

methods for developing software weren't getting diffused more readily. In your research,

you discovered that PSP is not getting diffused as readily as it might. What do you think

is the reason for that?

GG: There are a couple of things. We found that the degree to which the actual software

engineers are involved in the initial decision to adopt a new technology—for example, to

bring in PSP—has an impact on how widespread the use is. In general, we found that the

degree of developer involvement was on the low side. That could explain some of the

reasons why the use is not as widespread as we thought. In cases where the involvement

by the developer is greater, we tend to find a greater degree of use of PSP by that

developer.

Another issue is training. It turns out that training is an important indicator of sustained

use. So, the degree to which the developer was able to receive quality training and was

able to have sufficient time to work on the training-related assignments also seems to be

an important factor in the diffusion.

AH: Some survey respondents said that a weakness of PSP was that it was difficult to

understand how to adapt it to a requirements stage and also to a maintenance stage. Some

people felt that their projects were primarily maintenance or the evolution of an existing

system. They weren't quite sure how to adapt PSP to those types of environments.

GG: Another challenge was related to the whole issue of improvement in productivity.

Respondents indicated that PSP did seem to provide them with improvements in software

quality as far as decreasing the number of errors. However, we weren't able to show

productivity improvements through the results of the survey. Respondents said there was

a lack of immediate improvement in development productivity. I think part of that could

be due to the fact that we are talking about an innovation, so there is a learning curve.

The need for management persistence

BT: We've already heard that management support is critically important. What about management persistence—the idea that management must not only support the effort, but constantly encourage it.

AH: Regarding the amount of management support, or "champion support" as we titled it in our research—we found somewhat equivocal results. It's clear that champion support is essential to get an innovation off the ground. But we also found a need for developer involvement in the decisions to be made to bring that innovation in. Some people felt that the champion was too demanding: "You will do this no matter what." That took away from the developer's personal control or personal choice in the matter. One of our recommendations in the report was that there is a fine line to be drawn between the amount of champion support and the amount of user decision-making and involvement on the developer's side.

GG: On the issue of management support, you don't want to do too much, but at the same time, you don't want to do none at all. There's that fine line. As far as management support for training, we had good news. Software developers seemed to believe that management had provided enough time and funds for training. They felt that they had enough access to PSP experts and felt that they had a fairly good network of support. There is, however, the challenge that managers give them enough time to go to class but not enough time to effectively work through the assignments. There was also a comment about class sizes: classes might be more effective if they're smaller.

TH: In the academic environment, it's not so much the persistence of management as it is the persistence of the faculty throughout the curriculum. You can't simply teach it in one course and then expect students to acquire some sort of disciplined process that they're going to automatically apply in other areas.

RP: In our organization, PSP was not brought in by an engineer or a project manager who had to sell it up the management chain. It was brought in by the owner of the company and pushed out, which has some obvious advantages. We changed a lot of our management and organizational practices to be consistent with the use of PSP. We developed software engineering policy statements, stating that these things were going

to

be used. We integrated it into our performance-evaluation system at both the engineer and project-manager levels. Project managers have to report quarterly to senior management about which PSP practices are being applied to their projects. We provide support via PSP user group meetings at least once a quarter. Our organizational goals and objectives include developing high-quality components, which indirectly relates to using PSP in our development work. We decided that this was important to us, and that we were going to have to change our culture to accept it. We really took a serious look at it and changed things at the organizational level to support it all the way.

RC: When students write informal answers on a survey or a test, their writing looks totally different from when they write about something that they've thought about. It's as if they write with two different styles. What we're trying to do with PSP is to convince them that there's really one style of work. You don't have a formal working style and an informal working style. Your approach to the work is disciplined all the time.

Lessons learned from PSP and TSP training

BT: What are some lessons that you've learned from your experiences with PSP and TSP?

TH: I went through the PSP exercise one summer and it made me rather humble. I thought I was a lot better software developer than I was. But it did show me ways to improve and I saw improvement from the beginning. That's the sort of experience you hope all students and engineers who go through PSP training realize themselves. I have also changed my view of the way I teach programming. Software development and software engineering have changed dramatically. I hope I am much more of a coach and a mentor and a supporter than I am just a lecturer or a one-way facilitator. When I was at the SEI and sat through the engineers' course, I saw people who had been programming for 10 or 20 years who were struggling with some of these problems. The SEI does a very good job in their afternoon labs of helping people plan and learn PSP and develop quality software. I thought, "Here are some experienced engineers who are struggling with

how

to develop programs effectively." We just throw things at students sometimes—

especially at the undergraduate level—and expect them to develop their own effective

process. So, it changed my viewpoint of the way we've been educating our computer

scientists and software engineers.

RP: There are two lessons that revolve around the training itself and one that had an

impact on my associates and me. The industrial delivery of the course is a substantially

different professional training course than most of the courses you go to. This is not a 9-

to-3 type of course that meets for a handful of days for a light break away from the

normal workflow. This training is much more intense than what people are facing in their

normal daily efforts. There needs to be a strong understanding of the commitment and

time it takes to get through the training and an effort to make sure people have enough

time to get through the assignments and realize the full benefit of PSP. Then, afterwards,

it's important for people to understand that since this is not your typical professional

training course there needs to be some sort of vision. Why are we doing PSP? Are we just

throwing money at something, putting people into a class and hoping that productivity or

quality will improve afterwards? There really needs to be some sort of vision about

what's going to happen on that day after class is over. How do we see things changing?

How do we want to see things change? What kind of practices do we expect people to

start incorporating into their daily work? We should set the expectations of it and make

sure the infrastructure is in place to support that.

The third item reflects more on our organization. For about two years before we got

exposure to PSP, we were trying to get a CMM effort going and we went through some

early growing pains with that. When I went into the PSP class, I thought I had a good

notion of what the CMM was all about. However, the CMM is really big on telling you

what you should do and says very little in terms of how you should go about doing it.

Going through the PSP class was an eye-opening experience. I started understanding

exactly what all these things in the CMM were about and how they related to one

another. It really showed me exactly how to do these things. By the time the course was

over, I felt that it was a total eye-opening experience in terms of what our organization

was really trying to accomplish at a much higher level.

RC: For me, the principal lesson has been to approach much of the work I do in a manner similar to the way I learned to do things for PSP. I had never thought that I could work at some things in such a disciplined manner.

RP: The other lesson learned from our perspective is that it works. We recently reported that out of 148 components that were delivered to the customer over the past two years, 100 of them were defect-free. Our most recent project, which is now a month into customer use, has about 80,000 lines of code and the customer has reported only four defects. That's clearly superior quality in terms of what most organizations are able to produce.

Robert Cannon is professor of computer science at the University of South Carolina. During the 1998–99 academic year he was a visiting scientist in the Process Group of the SEI, working primarily on PSP and TSP. He has also been a visiting scientist with IBM Corp. and at the Center for Automation Research of the University of Maryland. A principal interest in recent years has been visualization of sedimentary processes, which he has performed jointly with a team of geological researchers. He is a member of the

Association for Computing Machinery and the Computer Society of the IEEE. He has been active in computer science accreditation and was president of the Computing Sciences Accreditation Board in 1998–99. He received the B.S. in mathematics from the University of North Carolina at Chapel Hill, the M.S. in mathematics from the University of Wisconsin at Madison, and the Ph.D. in computer science from the University of North Carolina at Chapel Hill.

Gina Green is an assistant professor of information systems at Baylor University. She holds a Ph.D. in management information systems from the University of South Florida, an M.S. in computer science from the University of Pennsylvania, and a B.S. in computer science from Southern University. She recently co-authored, with Alan Hevner, a special report published by the SEI titled *Perceived Control of Software Developers and Its Impact on the Successful Diffusion of Information Technology* (CMU/SEI-98-SR-013)

[URL to come]. During her professional career, Dr. Green worked as a systems engineer,

database administrator, and software development manager for IBM. Her academic

research interests include the diffusion of innovations, systems development methods,

project management, and complexity in computer-supported work. She is a member of

the Association for Computing Machinery, SIGMIS, and the Decision Sciences Institute.

Will Hayes is a member of the technical staff at the Software Engineering Institute. As a

research methodologist, Will participates in a diverse set of activities at the SEI including

empirical studies focused on the impact of technologies and methodologies in software

development organizations. As a member of the Software Engineering Measurement and

Analysis (SEMA) Initiative, Hayes provides consultation both within the SEI and to SEI

customers, in the area of measurement and empirical studies. He is an SEI-authorized

lead assessor and instructor of “Introduction to the CMM” and “Personal Software

Process” as well as “Implementing Goal-Driven Software Measurement.” He holds a

master of science in research methodology from the University of Pittsburgh, and a

bachelor of arts in psychology from Washington College in Chestertown, Md.

Alan R. Hevner is an eminent scholar and professor in the Information Systems and

Decision Sciences Department in the College of Business Administration at the

University of South Florida. He holds the Salomon Brothers/HRCP Chair of Distributed

Technology. Dr. Hevner's areas of research interest include software engineering,

distributed database systems, database system performance evaluation, and

telecommunications. He has more than 70 published research papers on these topics. He

has consulted for a number of organizations, including IBM, MCI, Honeywell, Cisco, and

AT&T. Dr. Hevner is a member of ACM, IEEE, AIS, and INFORMS.

Thomas B. Hilburn is a professor of computer science at Embry-Riddle Aeronautical

University. After graduation from college, Dr. Hilburn served in the U.S. Navy and did

work in the area of computer navigation systems. In 1973, he received his Ph.D. in

mathematics from Louisiana Tech University. Dr. Hilburn has taught mathematics,

computer science, and software engineering for 25 years and has received numerous

teaching awards. In 1991, Dr. Hilburn worked as a visiting scientist at the MITRE

Corp.

in the AERA 3 group. He headed a project investigating the use of pattern recognition to

automate analysis of airspace complexity. In 1993, Dr. Hilburn served as a senior

computer science lecturer for ICAO, in Cairo, Egypt, and taught Ada and software

engineering to Egyptian engineers engaged in developing ATC software. More recently,

Dr. Hilburn completed a sabbatical in the Process Program at the Software Engineering

Institute, where he worked on the transition of the Personal Software Process to industrial

and academic organizations. Dr. Hilburn's current interests include software processes,

formal specification techniques, and curriculum development. Dr. Hilburn is a member of

the ACM and the IEEE Computer Society, is the current software engineering category

editor for *ACM Reviews*, and serves on a number of professional and educational

committees for software engineering and computer science.

Robert J. Pauwels is the commercial training director at Advanced Information Services

Inc. (AIS). He has four and a half years of experience in Personal Software Process

education, consulting, and coaching. As commercial training director he is responsible for

working with commercial organizations in software process improvement and training.

He has 12 years of software development experience and has held other positions as

internal training director, project manager, and software engineer. He is an SEI

Authorized PSP instructor and delivers PSP courses to industrial organizations

internationally. In 1987 he received an undergraduate degree in computer science from

Rockford College. He is a steering committee member for the Chicago SPIN and is a

member of the IEEE Computer Society.

James E. Tomayko is a principal lecturer in the School of Computer Science (SCS) at

Carnegie Mellon University and a part-time senior member of the technical staff of the

Software Engineering Institute. He is the director of the Master of Software Engineering

Program in SCS. Tomayko directs the Software Development Studio for the MSE

Program. Previously he was leader of the Academic Education Project at the SEI, and

prior to that, he founded the software engineering graduate program at the Wichita State

University. He has worked in industry through employee, contract or consulting

relationships with NCR, NASA, Boeing Defense and Space Group, CarnegieWorks,

Xerox, the Westinghouse Energy Center, Keithley Instruments, and Mycro-Tek. He has

given seminars and lectures on software fault tolerance, software development management and software process improvement in the United States, Canada, Mexico, Argentina, Spain, Great Britain, and Brazil. Tomayko's courses on managing software development and overviews of software engineering are among the most widely distributed courses in the SEI Academic Series.

SEI Interactive, 06/99 page 17

http://interactive.sei.cmu.edu/Features/1999/June/Roundtable/Roundtable.jun99.htm

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.

i In order to use the Team Software Process, all team members must be trained in the Personal Software Process.

Delivering on the Promise of Process Improvement

NEWS AT SEI

This library item is related to the following area(s) of work:

Process Improvement

This article was originally published in News at SEI on: June 1, 1999

In 1995, the Software Engineering Institute introduced the Personal Software Process (PSP), a technology that asks software engineers to do their jobs an entirely different way. Through formal training, engineers learn to set personal goals for improvement, measure and analyze their work, and adjust their processes to meet their goals. They develop the ability to predict their performance and manage the quality of the work they produce. Organizations using PSP have reported significantly improved size and time-estimating accuracy, as well as reductions of 60–75% or more in the number of defects injected during development or found during unit test.

While the payoff is great in terms of improved quality and efficiency, so too is the required commitment. Often, PSP techniques only become ingrained after engineers use them consistently for several months. Also, PSP usually works best when entire teams are trained together to use the techniques. Recognizing these challenges, the SEI introduced an experimental version of the Team Software Process (TSP) in summer 1996 to address some of the issues that could prevent PSP from achieving complete success. Since then, several dozen TSP teams have been launched.

This article surveys some successful installations of PSP and TSP, and presents some lessons that organizations have learned along the way. As some of those lessons actually contradict each other, perhaps the first lesson is that PSP and TSP are evolving technologies that will work differently under different circumstances. (For an overview of PSP and TSP, please see the Background article in this issue of SEI Interactive.)

PSP takes hold at Boeing

The Boeing Company's John Vu has enjoyed remarkable success implementing the PSP, but not without learning some hard lessons along the way. Vu, a Boeing technical fellow and chief software engineer, is responsible for software engineering process

improvements company-wide. He says his early efforts to provide PSP training to individuals from around the company yielded few positive results.

“My number one lesson learned is that random training does not work,” Vu says. “If you have an open class, and you have different people from different groups, they may not be able to apply PSP/TSP successfully.” Vu has surveyed participants from the “random training” classes to determine whether they were still using PSP. “Only 10 percent said they were. You must focus on a specific project or program, build an infrastructure, and train everybody. Then it works very well.”

In successful PSP training at Boeing, Vu has concentrated on training groups of about 25 engineers over six months. A full-time “coach-instructor-mentor” works as a member of each group’s staff, and helps with training and collecting personal data and statistics, which are anonymously recorded and converted to trend lines. “Without a coach, you can’t maintain the focus and direction of the program the way you want it,” Vu says.

The groups have delivered impressive results. Defects have dropped by almost 75% over past software releases. Delivery time is up only one-quarter of 1%, but the groups are writing programs that are 2.4 times larger than previous programs, and three to four times more complex. “We do it faster and we reduce the defects by close to 75%,” Vu says. “To me, the data speaks for itself.”

Vu has experimented with different approaches for providing training, and has settled on a half day per week for six months. “Most engineers say one-half day per week is very reasonable. They complain that they don’t have enough time for the homework, so we are trying to give them another half day for homework.”

The two current programs are pilots, but Vu says he will soon have enough data to justify a company-wide PSP program.

Vu, who is an SEI resident affiliate and has worked closely with the SEI technical staff since 1988, says that despite Boeing's success, he has some concerns about PSP. “I believe you cannot use PSP/TSP before an organization has reached Level 3,” he says, referring to the SEI’s five-level Capability Maturity Model for Software. “At Level 3 or 4, the organization is more mature and has much better management support—and management support is critical.” In mature organizations, management understands that when a group of engineers, using PSP, provides a well-developed schedule for a project, the timetable is accurate, even though it may be longer than management would prefer. “If you don’t have a mature organization, managers will say, ‘I’ll give you one week, but not two.’ In more mature organizations, managers appreciate the data that engineers show them from PSP about the need for more time.”

Vu says he also believes that organizations that use PSP/TSP can accelerate to Levels 4 and 5 more quickly, and he is currently collecting data to support that belief.

Motorola applies PSP principles

Motorola first introduced the PSP approach in 1995 when a group of managers teamed up to explore ways to improve software engineering practices and achieve CMM Level 3, recalls John Pang, a staff engineer who was part of the original pilot project and has been working with several PSP projects since. Motorola hired an instructor from Embry-Riddle Aeronautical University who taught two courses, one for managers and one for engineers, using the textbook *Discipline for Software Engineering* by the SEI's Watts S. Humphrey.

Since that time, about 100 people at Motorola have been trained to use PSP, all in groups of coworkers. As at Boeing, Motorola experimented with the training schedule. Motorola has adapted the original schedule of 3 training days per week for 15 weeks, to 2 days per week for 10 weeks. The company does not provide trainees with time for homework, which they must do on their own time. The company acknowledges the engineers’ extra effort by throwing a party for the workers and their families at the end of the course.

Pang says that his organization achieved its goal of Level 3 assessment because of three factors:

1. strong support from senior management, including resources
2. <![endif]> the vision of the manager of the software engineering process group (SEPG), who set Level 3 certification as the overall goal
3. widespread training in the Personal Software Process, which helped the organization understand what it needed to do to reach Level 3

The SEPG leader "knew what he was doing. He knew how to talk to senior management, middle management, and the engineers, and how to steer them to his vision of CMM Level 3. But we could not have followed his vision for CMM Level 3 without PSP, because the engineers could have seen the effort as a burden, as extra work. PSP provided the education. It opened our eyes to the fact that process improvement is possible, and why things like requirements, planning, and inspection are important." Later, when the process group needed to institutionalize inspections, "We said, 'OK, that makes sense. Let's do it,'" Pang says.

The engineers at Motorola found, however, that they could not strictly adhere to the Watts Humphrey textbook. Rather, they needed to tailor the course to their own needs and organization, Pang says. "We took the principles, such as 'inspection is valuable' and 'fix defects as early as possible.' So we asked ourselves, 'How can we institutionalize inspections? How can we manage ourselves? How can we negotiate estimates and schedules?'" Groups also developed their own training exercises. Rather than using an example from the textbook, they built exercises around how to program functionality into a new line of pagers—an actual workplace project.

Pang declined to share specific data on improvements but, he says, "The engineers are much happier. They don't have to go into the factory because a bug shut down the line. We like our jobs better than we did five years ago."

Software engineers at AIS take to PSP

Advanced Information Services, based in Peoria, IL, became interested in software process improvement in 1992, when the owner of the consulting company read some of Watts Humphrey's writings on managing the software process. The owner asked Humphrey if the concepts could be applied to a small organization of only 35 employees, who worked on projects in teams of five or fewer engineers. At the time, Humphrey was developing the techniques that would become the Personal Software Process, and he met with AIS staff members to discuss his ideas. When the SEI taught its first instructor's training course for PSP in 1994, AIS sent two software engineers to become instructors.

Today, AIS employs some 150 software experts at its offices in Peoria and Evanston, IL, and at a wholly owned subsidiary in Chennai, India. The company provides software consulting services in a wide range of application areas such as embedded systems, information systems, and e-commerce, covering technologies including the Internet, client-server networks, and mainframes, and supporting programming languages from C to COBOL to Oracle.

AIS made an early commitment to PSP for the members of the firm's development group. "We knew that PSP was the way we were going to go for projects that we had control over," recalls Robert Pauwels, commercial training director for AIS. The firm ran a pilot project, then introduced PSP more broadly. Today, the PSP is a required training course for all AIS software professionals.

Pauwels says that he believes PSP was easier to implement at a consulting firm like AIS, as opposed to a manufacturing company, because AIS consultants are already used to some PSP principles, such as tracking time. "Being a consulting firm, we're used to time tracking for billing purposes. But detailed time tracking is extremely foreign to many organizations." AIS also had a defined process for software development, and practiced peer reviews to promote early defect detection.

For AIS today, the challenge is to quickly train new hires to use PSP. "We do a fair amount of recruiting from the universities in this area," Pauwels says. "We put recruits through the course as soon as we can. The ideas are still quite foreign to them, compared with what they've just invested four years of their lives in at the university.

Most are surprised at what we're asking them to do. But it is entrenched in our culture and institutionalized, and that helps break down the barriers of resistance that they often have."

As at other organizations, PSP has shown powerful results at AIS. "One of our goals from an organizational standpoint is to produce software modules with zero defects, and we are leveraging the PSP skills to do so." For example, on their most recent projects, 100 out of 148 modules had zero defects from integration test forward. In another example, AIS completed an 80,000-line Visual Basic project. The delivered defect ratio was 0.05 defects per 1,000 lines of code—a small fraction compared to the number of defects that would be found on a similar project for which PSP was not used.

Pauwels adds that PSP, and also CMM training, have greatly increased the speed at which AIS completes projects. "It used to be that our projects would take months to get through system test. We've been able to squeeze that down to days now by putting a focus on defect detection throughout the development process. In the early days, we had to wait until test to get our first indication of product quality. By coupling PSP and CMM together, early defect detection has become ingrained in our culture."

Early observations of TSP

Since the SEI introduced the Team Software Process in summer 1996, several dozen TSP projects have been launched with commercial and government organizations. From those experiences, the SEI's James Over has collected some "war stories" about TSP's early implementation.

In one example, a team of nine software engineers and five hardware engineers were assigned the delivery of a critically needed hardware–software product in nine months or less. The team defined its development process, estimated the size of the job, developed an overall plan, produced detailed next-phase plans, balanced those plans, and assessed project risks and problems. This effort called for an 18-month delivery schedule. Under strong management pressure, the team explained its plan and justified the effort required. Management eventually accepted the plan and the team finished six weeks ahead of schedule.

On another team, defect ratios have dropped dramatically. Before the organization instituted the TSP approach, a project with 9,500 lines of code had an average of five defects per 1,000 lines. After TSP training, the team completed a project with 89,900 lines of code, which had .02 defects per 1,000 lines. That is an improvement rate of 250 times.

Where successfully implemented, the TSP approach "provides teams with detailed plans, the data to justify their commitments, and the conviction to defend their plans," Over says. "In every case, management has accepted the team's committed dates. In all cases where the team has followed the process, they have met their dates." In addition, Over says, "When engineers use the TSP, they produce very high-quality products. This is because they measure and track quality, apply quality methods, and have quantified quality goals."

Still, not every TSP implementation has been a success. Since 1996, two TSP projects have failed, both because of major management shakeups that resulted in pressure to cut costs and widespread disruptions of engineering staff. Also, three projects have had only "so-so" results. Of those, none included management training and management did not monitor the teams' performance. The engineers involved eventually stopped gathering and using TSP data and the projects did not follow the TSP process. Nonetheless, exposure to TSP made a difference: the projects all delivered working products, finished close to on time, and appeared to produce improved quality.

Over also considers some projects to be "shaky." "The shaky projects have many part-time engineers, untrained management, or no coaching support," he says.

TSP success at Hill Air Force Base

One of the most successful TSP implementations to date is at the Hill Air Force Base (AFB) software group, which develops avionics and support software for the U.S. Air

Force. The group sent several engineers to the SEI's PSP instructor authorization program. In a recently completed pilot project of 25,820 lines of code using the PSP and TSP approaches, the project team delivered the product a month ahead of the originally committed date, at almost exactly the planned cost. During system and operational tests, the team found only one high-priority defect and reduced testing time from 22% to 2.7% of the project schedule. This product is currently in acceptance testing, and the customer has found no defects to date.

"TSP gives you incredible insight into project performance," says one member of the Hill AFB software group. "You can really see what needs to be done...I saw this group go from setting 'square-filler' goals to setting real, achievable goals, and now I see them achieving those goals."

"TSP creates more group involvement," says another Hill AFB engineer. "Everyone feels like they're more part of a group instead of a cog in the wheel. It forces team coordination to talk about and solve problems."

About the author

Bill Thomas is the editor in chief of *SEI Interactive*. He is a senior writer/editor and a member of the SEI's technical staff on the Technical Communication team, where his duties include primary responsibility for the documentation of the SEI's Networked Systems Survivability Program. His previous career includes seven years as director of publications for Carnegie Mellon University's graduate business school. He has also worked as an account manager for a public relations agency, where he wrote product literature and technical documentation for Eastman Kodak's Business Imaging Systems Division. Earlier in his career he spent six years as a business writer for newspapers and magazines. He holds a bachelor of science degree in journalism from Ohio University and a master of design degree from Carnegie Mellon University.

The Net Effects of MP3

NEWS AT SEI

Author

Scott R. Tilley (Florida Institute of Technoloy)

This article was originally published in News at SEI on: June 1, 1999

Listen closely and you'll hear yet another Net-driven revolution in progress: direct-to-the-consumer digital music. The music is stored in a format called MP3, which provides near-CD quality sound using a fraction of the disk space previously required. As with other phenomena popularized by an energetic Internet community, MP3 is shaking the very foundations of the established industry.

Consumer demand and the open source movement

The Internet's aptitude for dramatically altering existing business models has become readily apparent of late. The open source movement, as exemplified by the Linux operating system and the Apache Web server, is causing a measurable change in the attitudes of established corporations. What is truly amazing is that these changes are being driven by consumer demand for products and services that companies didn't know existed, or are reluctant to pursue.

A prime example of a new market trend that many staid companies are hesitant to embrace is the MP3 music phenomenon. Many people already enjoy streaming audio, such as that provided by Web-casting radio stations in the RealAudio format from RealNetworks, Inc. Now a new technology is taking center stage, a digital audio format called MP3 that provides near-CD quality recording and playback in a highly compressed format.

The effects of Internet-driven, bottom-up events such as MP3 are profound and far-reaching. In this column, I focus on the net effects of MP3 on several entities: the recording industry, the companies providing new media players such as RealPlayer and QuickTime, and users. First, let's look at what makes MP3 unique.

MP3

The sometimes low quality of streaming audio programs, which often sound like distant AM radio broadcasts, has increased interest in several new digital audio

standards. MP3 is the most popular digital audio format on the Internet. The term MP3 refers to MPEG 1 layer 3 audio. MPEG stands for Motion Picture Experts Group, an international standards body. MP3 has become very popular because it is a non-proprietary open standard that offers compressed music with near-CD quality sound. It is not streaming, which refers to a transmission format that allows the user to experience audio and video as it is downloaded from a network in real time. But because of the high compression ratios possible, downloads are not nearly as onerous as they are with other audio formats, such as Windows' WAV files.

Like other digital compression techniques, MP3 uses a "perceptual coding" approach to reduce the size of the music file. It works by identifying portions of the audio that the human ear cannot hear, and removing them from the sample. This allows the MP3 encoder to produce a much more compressed version of the recording, but with sound fidelity nearly indistinguishable from the original.

The degree of compression offered by MP3 is variable; the more compression, the smaller the file size, but the lower the audio quality. However, even at high compression ratios of 18:1, the resultant audio file produces sound that is comparable to a CD. A four-minute song on a CD takes about 40 megabytes of storage. When converted to MP3, the file is about 2.6 megabytes. At a less aggressive 12:1 compression ratio, which is true CD quality, the MP3 file takes about 4 megabytes.

The effects on the recording industry

The group with perhaps the most to lose from the propagation of MP3 music is the established recording industry. One of the reasons the MP3 format is such a worry to the industry is the proliferation of pirated music on the Internet, seemingly free for the downloading. Digital copies can be made with no loss of fidelity and most MP3 players do not provide any copyright-protection mechanisms.

Recording companies fear this will cause people to download music without paying royalties, thereby undercutting their entire business model. However, similar concerns were voiced when the tape recorder and the VCR initially appeared. Since their introduction, music sales and video sales and rentals have grown tremendously.

Nevertheless, the Internet-oriented nature of the MP3 format may make the current situation a little different from what occurred with the tape recorder or the VCR. For example, unsigned artists are setting up their own e-commerce Web sites and offering their music for a small fee, bypassing the traditional recording industry's established methods for distributing artists' work. As with all revolutions, such dramatic changes tend to make some people very nervous—especially those with a stake in maintaining the status quo.

The effects on new media players

There is little doubt that MP3 has the potential to greatly affect the established recording industry. But what about the effects on the so-called "new media" establishment? Internet-oriented products like RealNetworks' RealPlayer, Microsoft's Media Player, and Apple Computer's QuickTime all provide recording and playback of digital audio and video. They have all added support for MP3 to their products, and have been quicker to react to the MP3 phenomenon than the old-guard recording industry has, but that may be because they don't have as much to lose.

In the streaming media arena, there are several competing formats vying for dominance. RealNetworks is the undisputed leader in streaming-media technology from a market-share perspective. Its free RealPlayer program is installed on most personal computers, allowing people to experience Web-casting radio stations and live video feeds. As with most things Internet, the technology is evolving so rapidly that consumers and companies can barely keep up. For example, Microsoft released its new MS Audio 4 suite a few weeks ago. Apple Computer released QuickTime 4, a streaming version of its popular movie player at about the same time. Technology discussions aside, neither of these two alternatives has had nearly the impact as RealPlayer.

However, RealNetworks' focus has so far been on streaming media, which has the advantage of supporting live broadcasts, but the disadvantage of suffering from the vagaries of the Internet. For example, it must compensate for lost packets and network

outages.

Consumer demand for MP3 has forced RealNetworks, Microsoft, and Apple to adapt their media players to support MP3 in addition to their proprietary formats. It is clear that they too are being affected by the encroachment of MP3 on their products.

The effects on users

As I type this, I am using a free MP3 player to listen to music that I "ripped" from my own CD collection. The sound quality is such that most people would not know that the source is my hard drive, not the CD player. I've tried several different programs for encoding and playing MP3 files. Most of them are a little rough around the edges, but RealNetworks' recently released RealJukebox product may change that by bringing MP3 to the masses.

Perhaps more significantly, the recent introduction of the Rio portable MP3 player from Diamond Multimedia Systems has caused quite a stir. Weighing just over two ounces, the Rio holds about one hour's worth of near-CD quality music. It has no moving parts, lasts twelve hours on a single AA battery, and connects to your computer to download MP3-encoded music using a simple cable. It also has an expansion slot that accepts flash memory cards, the type used in digital cameras, to hold more music.

Diamond is not the only manufacturer of players that is hopping onto the MP3 bandwagon. Samsung, Creative Labs, and Thomson Multimedia are just three consumer electronics companies with MP3 players in the pipeline. Creative shipped its Nomad product this month, and Thomson plans to ship its Lyra product later this summer. All are second-generation MP3 players that include enhancements such as voice recording, FM radio reception, and support for other digital audio formats.

How will this digital downloading affect the current preferred music format, the compact disc? It might be wise to consider the fate of the phonographic record player, which is all but extinct. In this net-centric music scenario, CDs are anachronisms. Why have CDs at all, when the music can be downloaded in digital format directly from the Internet?

Does this mean you have to junk your CD collection and start investing in memory cards instead? Not entirely. But the adoption of MP3, or even better compression formats to come, will affect how you experience music in the future. It may very well be that all music becomes digital, downloaded, and dramatically better. The net effects of MP3 on users will be positive and beneficial.

About the author

Scott Tilley is a visiting scientist with the Software Engineering Institute at Carnegie Mellon University, an assistant professor in the Department of Computer Science at the University of California, Riverside, and principal of S.R. Tilley & Associates, an information technology consulting boutique.

412-268-5800

Evaluation of COTS Products: Some Thoughts on the Process

NEWS AT SEI

Author

David J. Carney

This library item is related to the following area(s) of work:

System of Systems

This article was originally published in News at SEI on: September 1, 1998

This column will be devoted to discussions about the use of commercial off-the-shelf software (generally called "COTS") in defense and government systems. In the first two columns, this one and the succeeding one, I will speculate about the process of evaluating commercial software: supposing that we had the technical apparatus to measure the key attributes of a software component (which is at least debatable), how would we go about doing it? When would it be done? Who would get the job? These are questions that fall under the heading of process, and these considerations are as significant for COTS-based development as they are for more traditional ways of building systems.

Introduction

We are presently witnessing a widespread movement by government and industry organizations toward use of COTS products either as stand-alone solutions or as components in complex, heterogeneous systems. This trend results from the realization that using preexisting software products can be a means of lowering development costs, shortening the time of system development, or simply maintaining currency with the rapid changes in software technology that are taking place today.

However, in choosing to make use of commercial components, an organization must immediately deal with the problem of assessing or evaluating these products. The technical literature has examples that describe some specific techniques for assessing a commercial product, its attributes, and its fitness for use in a given context.

Related Links

Training

Migrating Legacy Systems to SOA Environments - eLearning

Service-Oriented Architecture: Best Practices for Successful Adoption

See more related courses »

In addition to considering techniques in themselves, however, we also observe the need to define some of the more general process-related issues that arise when evaluating COTS products. For example, whose job is it to do this? How do the traditional notions of evaluation differ from COTS evaluation? What new activities might be implied when COTS products are under evaluation? What implications exist for the relationships between these activities, or for their sequence? To begin to answer some of these questions, I will consider some preliminary notions about the process of COTS evaluation. I will examine and make some suggestions about the nature of evaluation, the overall process, and the major roles that must be filled to accomplish it. In my next column, I will examine how these ideas might be instantiated in an actual evaluation exercise.

The Meaning of "COTS Evaluation"

At the outset, we first must constrain the domain of discourse. While evaluation activities are pertinent to any selection of a COTS product, they are especially important when the product will be a component in a complex, heterogeneous system and when the constraints native to the product and its vendor must be harmonized with the constraints of the system that incorporates it. Most of the descriptions found herein are based on the assumption of such a system.

Even in the restricted context of complex, heterogeneous systems, the term *evaluation* is used with a great many meanings by various people. One common understanding of *evaluation* makes it roughly synonymous with *acceptance testing*. Another common (though quite different) understanding of *evaluation* is that it refers to assessing software through such mechanisms as benchmark tests. Still another understanding of the term considers that evaluation is the activity performed at the close of a project, to determine its success and to capture the good and bad lessons that have been learned. None of these definitions can claim to be *the* correct one, but all are in common use, and all have some measure of validity.

In our experience, this diversity of meaning and understanding often gives rise to confusion and frustration. The way to alleviate this problem is precision. The very term *COTS*, for instance, centers our interest on commercial products. But even that is not sufficiently precise, since our major interest is evaluation of commercial products *for the purpose of deciding whether to select one for use*. Thus, COTS evaluation in this view is a *decision aid*, a notion that is quite different from evaluation as acceptance testing. The distinction is critical, and I intend the term *evaluation* to have only this restricted meaning. For the remainder of this column, therefore, I claim a very precise focus: given a set of commercial products, and a need to decide whether to select one for use, *evaluation* is the overall term I use for the broad collection of activities performed toward that end.

The Permeative Nature of COTS Evaluation

One consequence of this view is that COTS evaluation is *not* a cleanly separable action, but is more permeative: it exists in multiple forms and at subtly different levels. For example, let us leave the domain of software and consider an everyday model of "off-the-shelf evaluation," where a consumer magazine is consulted to help choose an automobile. Presumably, the overall process will be to (1) assemble some list of candidate cars, (2) apply some evaluation techniques using the available information found in the magazine, and (3) make a selection. What I claim as the "permeative" characteristic is that although evaluation appears to be happening only in the second step, it is also happening, with varying degrees of explicitness, throughout (1) and (3); in essence, evaluation activities pervade the entire process.

Suppose, for instance, that the fictional car buyer has consulted the issue containing reports on the latest foreign cars, seeking guidance on the new imports: We might think that this step is somehow "pre-evaluative," and occurs in advance of any real evaluation process. But consider how many selections (e.g., decisions about inclusion and omission) have already occurred. Used cars have been rejected. Domestic cars have been rejected. Both of these decisions have presumably occurred through some sort of evaluation process. And, unless the fictional purchaser has unlimited funds, there are implicit criteria that exist—cost, fuel economy, safety features, and so forth. Some of these constraints and criteria have been included, and others excluded, and

they are in some way prioritized. All of these decisions are, whether implicitly or explicitly, based on some form of evaluation.

Thoughts About the Overall Process

The phrase *overall process* is somewhat misleading, since while I am concerned with the process aspect of COTS evaluation, my intention is not to define a specific process to accomplish this. On the contrary, as with any important process consideration, an actual evaluation process will depend on a large number of variables that are particular to the organization performing an evaluation--the kind of problem, the specific needs of users, the products under examination, and so forth.

At the most abstract level, there are three large-scale tasks that are involved when COTS products are evaluated:

1. Plan the evaluation.
2. Design the evaluation instrument.
3. Apply the evaluation instrument.

(I use the phrase *evaluation instrument* to refer to the collection of constraints, strategies, and techniques that cooperatively are the mechanism for evaluation. In a very informal evaluation, this "instrument" may be nothing other than an individual's intuition. In a formal, costly, and complex evaluation of life-critical software, this "instrument" will likely have extensive documentation and be the object of considerable debate and refinement.)

Each of these three broad tasks has within it a collection of activities that are performed. This common collection of activities will usually be carried out for any specific process that is used. Their order of execution, the scope of each activity, and even whether all of them are always performed will vary, depending on circumstance. In effect, these activities are the primitive building blocks of an evaluation process, and the three large-scale tasks provide generalized areas within which the specific activities are performed. These constituent activities might be something like the following:

Plan the evaluation.

- Define the problem.
- Define the outcomes of the evaluation.
- Assess the decision risk.
- Identify the decision maker.
- Identify resources.
- Identify the stakeholders.
- Identify the alternatives.
- Assess the nature of the evaluation context.

Design the evaluation instrument.

- Specify the evaluation criteria.
- Build a priority structure.
- Define the assessment approach.
- Select an aggregation technique.
- Select assessment techniques.

Apply the evaluation instrument.

- Obtain products.
- Build a measurement infrastructure.
- Perform assessment.
- Aggregate data.

Form recommendations.

The Roles That Must Be Filled

If we are concerned with tasks and process steps, it is also true that we must consider who will perform these. There are three essential roles in the COTS evaluation process:

1. decision maker
2. analyst
3. stakeholders

The decision maker is the person (or persons) who has both the authority and the need to select a COTS product. While it is common for there to be a single decision maker, it is equally common for this role to be diffuse, and performed either by a committee or by some other distributed entity. The analyst is the person (or persons) who designs and executes the various activities that constitute a COTS product evaluation. The scope of this role is related to the scope of the evaluation. For a large or complex evaluation process, there will generally be some notion of a "lead analyst" whose work is primarily conceptual and analytical, with the actual hands-on tasks being done by other analysts. The stakeholders are all of the persons who share a problem or need, and who will benefit in some manner if a commercial product can alleviate that problem or need.

These rather disconnected thoughts have not arrived at any sweeping conclusions. However, by identifying in the abstract the key activities and roles that are involved with COTS evaluation, we can now begin to construct portions of actual processes and eventually arrive at some useful understanding of "how to do" COTS evaluation. In the next column, I will take some existing evaluation practices and examine them in comparison to the activities and roles described here. Stay tuned.

About the Author

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

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

Pathways to Process Maturity: The Personal Software Process and Team Software Process

NEWS AT SEI

Author

Watts S. Humphrey

This library item is related to the following area(s) of work:

Process Improvement

TSP

This article was originally published in News at SEI on: December 1, 1999

Editor's Note: The following article is a condensed version of a three-part series that appeared in successive issues of CrossTalk from February to April 1998. Those issues are available online at http://stsc.hill.af.mil/CrossTalk.

Although the Capability Maturity Model (CMM) provides a powerful improvement framework, its focus is necessarily on "what" organizations should do and not "how" they should do it. This is a direct result of the CMM's original motivation to support the Department of Defense acquisition community. We knew management should set goals for their software work but we also knew that there were many ways to accomplish these goals. Above all, we knew no one was smart enough to define how to manage all software organizations. We thus kept the CMM focus on goals, with only generalized examples of the practices the goals implied.

As organizations used the CMM, many had trouble applying the CMM principles. In small groups, for example, it is not generally possible to have dedicated process specialists, so every engineer must participate at least part time in process improvement. We kept describing to engineers what they ought to do and they kept asking us how to do it. Not only did this imply a need for much greater process detail, it also required that we deal more explicitly with the real practices of development engineers. We needed to show them precisely how to apply the CMM process principles.

Related Links

News

Heartbleed: Analysis, Thoughts, and Actions

TSP Symposium 2014 Goes Beyond Methodology to Focus on Software Quality

See more related news »

Training

See more related courses »

Events

Team Software Process (TSP) Symposium 2014

Nov 3 - 6

Improvement requires change, and changing the behavior of software engineers is a nontrivial problem. The reasons for this explain why process improvement is difficult and illustrate the logic behind the Personal Software ProcessSM (PSPSM).

A question of conviction

Software engineers develop their personal practices when they first learn to write programs. Since they are given little or no professional guidance on how to do the work, most engineers start off with exceedingly poor personal practices. As they gain experience, some engineers may change and improve their practices, but many do not. In general, the highly varied ways in which individual software engineers work are rarely based on a sound analysis of available methods and practices.

Engineers are understandably skeptical about changes to their work habits; although they may be willing to make a few minor changes, they will generally stick fairly closely to what has worked for them in the past until they are convinced a new method will be more effective. This, however, is a chicken-and-egg problem: engineers only believe new methods work after they use them and see the results, but they will not use the methods until they believe they work.

The Personal Software Process

Given all this, how could we possibly convince engineers that a new method would work for them? The only way we could think of to change their behavior was with a major intervention. We had to directly expose the engineers to the new way of working. We thus decided to remove them from their day-to-day environment and put them through a rigorous training course. As shown in Figure 1, the engineers follow prescribed methods, represented as levels PSP0 through PSP3, and write a defined set of 10 programming exercises and five reports1. With each exercise, they are gradually introduced to various advanced software engineering methods. By measuring their own performance, the engineers can see the effect of these methods on their work.

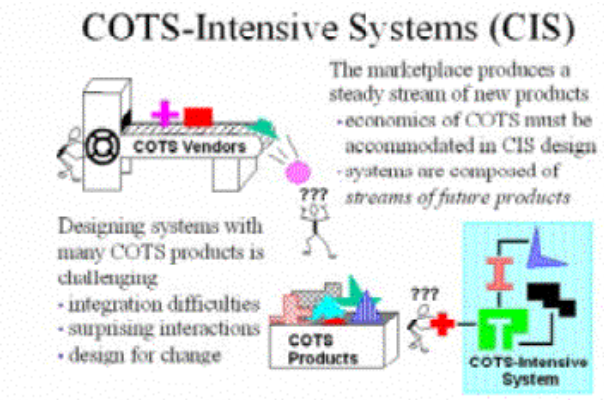

Figure 1: The PSP process evolution

Figures 2 through 4 show some of the benefits engineers experience2 3. Figure 2 shows an improvement from a 55 percent estimating error to a 27 percent error or a factor of about two. As shown in Figure 3, the improvement in compile and test defects is most dramatic. From PSP0 to PSP3, the engineers' compile and test defects dropped from 110 defects per 1,000 lines of code (KLOC) to 20 defects per KLOC, or over five times. Figure 4 shows that even with their greatly improved planning and quality performance, the engineers' lines of code productivity was more or less constant.

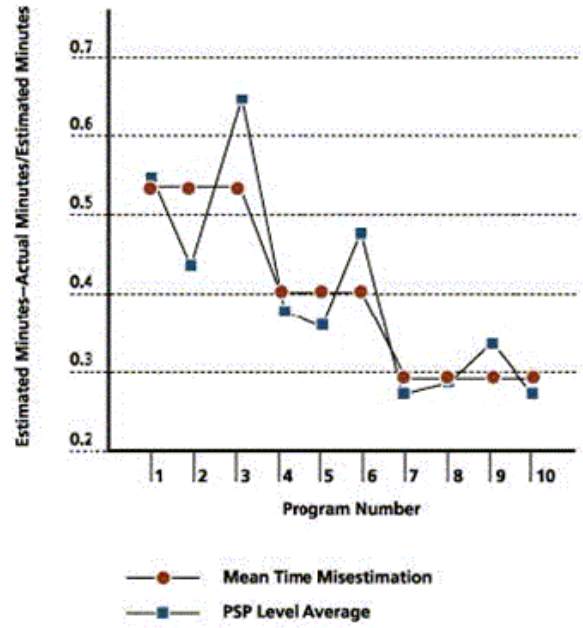

Figure 2: Effort estimation results

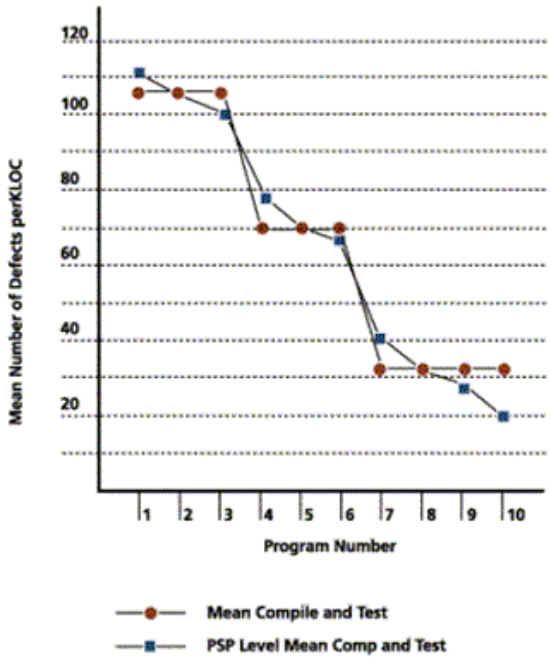

Figure 3: Quality results

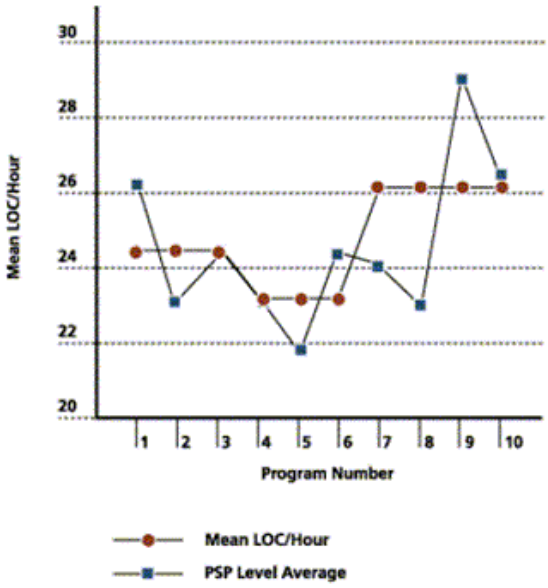

Figure 4: Productivity results

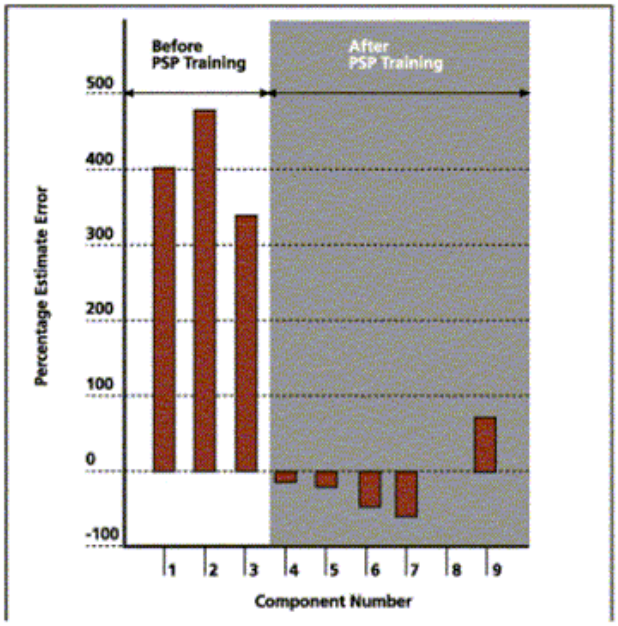

Figure 5: Schedule estimating error

Industrial results with the PSP

A growing number of organizations are using the PSP, such as Baan, Boeing, Motorola, and Teradyne. Data from some early users clearly demonstrate the benefits of PSP training4. Figure 5 shows data from a team at Advanced Information Services (AIS) in Peoria, Ill. Team members were PSP trained in the middle of their project. The three bars on the left of the chart show the engineers' time estimates for the weeks it would take them to develop the first three components. For Component 1, for example, the original estimate was four weeks, but the job took 20 weeks. Their average estimating error was 394 percent. After PSP training, these same engineers completed the remaining six components. As shown on the right, their average estimating error was -10.6 percent. The original estimate for Component 8, for example, was 14 weeks and the work was completed in 14 weeks.

Table 1 shows acceptance test data on products from one group of AIS engineers. Before PSP training, they had a substantial number of acceptance test defects and their

products were uniformly late. After PSP training, the next product was nearly on schedule, and it had only one acceptance test defect. Table 2 shows the savings in system testing time for nine PSP projects. At the top of the chart, system test time is shown for several products that were completed before PSP training. At the bottom, system test time is shown for products the same AIS engineers completed after PSP training. Note that A1 and A2 are two parts of the same product, so testing for them was done together in one and one-half months.

Not Using PSP	KLOC	Months Late	Acceptance Test Defects
1	24.6	9	N/A
2	20.8	4	168
3	19.9	3	21
4	13.4	8+	53
5	4.5	8+	25
Using PSP	**KLOC**	**Months Late**	**Acceptance Test Defects**
1	22.9	1	1

Table 1: Acceptance test improvement

System test time before PSP training

Project	Size	Test Time
A1	15,800 LOC	1.5 months
C	19 requirements	3 test cycles
D	30 requirements	2 months
H	30 requirements	2 months

System test time after PSP training

Project	Size	Test Time
A2	11,700 LOC	1.5 months
B	24 requirements	5 days
E	2,300 LOC	2 days
F	1,400 LOC	4 days
G	6,200 LOC	4 days
I	13,300 LOC	2 days

Table 2: System test time savings

Introducing the PSP

Although the PSP can be introduced quickly, it must also be done properly. First, the engineers need to be trained by a qualified PSP instructor. The SEI trains and authorizes PSP instructors and provides limited on-site PSP training. There is also a growing number of SEI-trained PSP instructors who offer commercial PSP training.

The second important step in PSP introduction is to train in groups or teams. When organizations ask for volunteers for PSP training, they get a sparse sprinkling of PSP skills that will generally have no impact on the performance of any project. Third, effective PSP introduction requires strong management support. This, in turn, requires that management understand the PSP, know how to support their workers once they are trained, and regularly monitor their performance. Without proper management attention, many engineers gradually slip back into their old habits. The problem is that software engineers, like most professionals, find it difficult to consistently do disciplined work when nobody notices or cares. Software engineers need regular coaching and support to sustain high levels of personal performance.

The final issue is that even when a team of engineers are all PSP trained and properly supported, they still have to figure out how to combine their personal processes into an overall team process. We have found this to be a problem even at higher CMM levels. These are the reasons we are developing the Team Software ProcessSM (TSPSM).

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 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:

- 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?
- 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 6, 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 four-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 the second and each subsequent phase. These launches are not training; they are part of the project.

Figure 6: TSP structure

The TSP launch process

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

- Review project objectives with management.
- Establish team roles.
- Agree on and document the team's goals.
- Produce an overall development strategy.
- Define the team's development process.
- Plan for the needed support facilities.
- Make a development plan for the entire project.
- Make a quality plan and set quality targets.
- 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.
- Hold a launch postmortem.

In the final launch step, the team reviews its plans and the project's key risks with management. Once the project starts, the team conducts weekly team meetings and periodically reports its status to management and to the customer. In the four-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, the early results are encouraging. One team at Embry-Riddle Aeronautical University removed more than 99 percent of development defects before system test entry. Their defect-removal profile is shown in Figure 7.

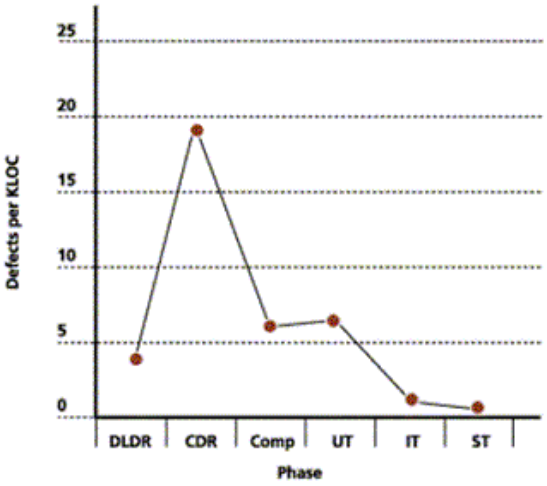

Figure 7: Defects per thousand lines of code (KLOC) removed by phase

Another team at Hill Air Force Base reduced testing time by eight times and more than doubled their productivity. The team's customer has since found no defects in using the product. [For more information about this case, see "Using the TSP on the TaskView Project" in the February 1999 issue of CrossTalk.] TSP teams also gather the data they need to analyze component quality before integration and system testing.

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 management's edicted schedule and is therefore unacceptable. As a result, they must work without the guidance of an orderly plan. Under these conditions, the team 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 teams 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. 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 3, 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.

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		
3 Defined	Engineering Process	Organization Process Focus	X	X
		Organization Process Definition	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	X
		Software Project Planning	X	X
		Software Project Tracking	X	X
		Software Quality Assurance		X
		Software Configuration Management		X
		Software Subcontract Management		

Table 3: PSP and TSP coverage of CMM key process areas

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.

About the author

Watts S. Humphrey founded the Software Process Program at the SEI. He is a fellow of the institute and is a research scientist on its staff. From 1959 to 1986, he was associated with IBM Corporation, where he was director of programming quality and process. His publications include many technical papers and seven books. His most recent books are: Managing the Software Process (1989), A Discipline for Software Engineering (1995), Managing Technical People (1996), Introduction to the Personal Software Process (1997), and Introduction to the Team Software Process (in press). He holds five U.S. patents. He is a member of the Association for Computing Machinery, a fellow of the Institute for Electrical and Electronics Engineers, and a past member of the Malcolm Baldridge National Quality Award Board of Examiners. He holds a BS in physics from the University of Chicago, an MS in physics from the Illinois Institute of Technology, an MBA from the University of Chicago, and has been awarded an honorary Ph.D. in software engineering from Embry-Riddle Aeronautical University.

1 Humphrey, W.S., A Discipline for Software Engineering, Addison-Wesley, Reading, Mass., 1995.

2 Hayes, Will, "The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers," CMU/SEI-97-TR-001.

3 Humphrey, W.S., "Using a Defined and Measured Personal Software Process," *IEEE Software*, May 1996.

4 Ferguson, Pat, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," *IEEE Computer*, Vol. 30, No. 5, May 1997, pp. 24-31.

For more information

Contact Us

info@sei.cmu.edu

412-268-5800