Carnegie Mellon
Software Engineering Institute

SEI interactive

Volume 3 . Issue 1 . March 2000

Full Issue

In This Issue:

Feature

Software Engineering Measurement and Analysis

Open Source Software

Columns

An Architectural Approach to Software Cost Modeling

COTS and Risk: Some Thoughts on How They Connect

Survivability Blends Computer Security With Business Risk Management

Justifying a Process Improvement Proposal

http://interactive.sei.cmu.edu

SEI Interactive

Volume 3 . Issue 1 . March 2000

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.

Table of Contents

Volume 3, Issue 1 . March 2000

About the SEI **3**

Background
Software Engineering Measurement and Analysis **10**

Spotlight
A Practical Approach to Improving Pilots **16**

Roundtable
A Discussion of Open Source Software **20**

Links
Resources on the Web **34**

The Architect
An Architectural Approach to Software Cost Modeling **38**

COTS Spot
COTS and Risk: Some Thoughts on How They Connect **45**

Security Matters
Survivability Blends Computer Security With Business Risk Management **51**

Watts New?
Justifying a Process Improvement Proposal **57**

About the SEI

Mission

The SEI mission is to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software. The SEI expects to accomplish this mission by promoting the evolution of software engineering from an ad hoc, labor-intensive activity to a discipline that is well managed and supported by technology.

SEI Work

The SEI program of work is grouped into two principal areas:

- Software Engineering Management Practices
- Software Engineering Technical Practices

Within these broad areas of work, the SEI has defined specific initiatives that address pervasive and significant issues impeding the ability of organizations to acquire, build, and evolve software-intensive systems predictably on time, within expected cost, and with expected functionality. Visit the initiative page on the SEI Web site [http://www.sei.cmu.edu] for more information.

From the Director

Watts New? The Entire Collection, That's What

One of the great strengths of *SEI Interactive* is its columns. Since June 1998, the columns in *SEI Interactive* have contributed to the bodies of knowledge on software architecture (The Architect), commercial off-the-shelf software (COTS Spot), computer security (Security Matters), and software engineering process improvement (Watts New?).

In the case of Watts New?, columnist Watts Humphrey--perhaps the best-known member of the SEI technical staff and the founder of the SEI's Software Process Program--has taken readers on a process-improvement journey, step by step. Watts made his intentions for the column clear in his first introduction, for "Why Does Software Work Take So Long?":

> *"In writing this column, I plan to discuss various topics of interest to software professionals and managers. In general, I will write about issues related to engineering productivity, quality, and overall effectiveness. Occasionally, I will digress to write about a current hot item, but generally I will be pushing the process improvement agenda. Because my principal interest these days is getting organizations started using the Personal Software Process℠ (PSP℠) and Team Software Process℠ (TSP℠), readers should know that a not-so-hidden agenda will be to convince them to explore and ultimately adopt these technologies."*

His columns have explored

- the problem of setting impossible dates for project completion ("Your Date or Mine?")
- planning as a team, using TSP ("Making Team Plans")
- the importance of removing software defects ("Bugs or Defects?")
- applying discipline to software development ("Doing Disciplined Work")
- approaching managers about a process improvement effort ("Getting Management Support for Process Improvement") and making a persuasive case for implementing it ("Making the Strategic Case for Process Improvement")

With this issue, Watts presents an example of a process improvement proposal -- complete with the numbers to back it up ("Justifying a Process Improvement Proposal"). We don't want to give away the ending but the five-year savings are about $10 million and the five-year return on investment is 683%!

We think these columns are an important contribution to the software engineering literature. As such, we have collected them into a complete set, which you can download as a PDF file. We hope that having *The Watts New? Collection* in one volume will make it easier to implement software process improvement in your organization.

Honored in India

Our publication of *The Watts New? Collection* comes on the heels of a major honor for Watts, and the SEI. On Feb. 23, 2000, Watts attended the inauguration of the Watts Humphrey Software Quality Institute in Chennai, India. In conjunction with his visit, one of the computer newspapers in India published a special supplement about the event.

In addition, the March 1, 2000, issue of *Business Week* has a Newsmaker Q&A interview with Watts, titled "The Guru of Zero-Defect Software Speaks Out." And Watts was recently chosen as one of the ten influential men and women of software, according to the managing editor of *CrossTalk* magazine, in an article published in the December 1999 issue.

Sincerely,
Stephen E. Cross
Director, Software Engineering Institute

From the Editor
Welcome to SEI Interactive

Welcome to the eighth installment of *SEI Interactive.*

With this issue we've made a subtle change to the publication. Rather than focus exclusively on one subject, as we have for all previous issues, we've tackled two subjects. Our Background and Spotlight articles address software engineering measurement and analysis, while our Roundtable focuses on open source software.

Cost overruns, late deliveries, and poor quality are problems that often plague organizations that develop and acquire software-intensive systems in both the defense and commercial sectors. The SEI has developed numerous methods to address those problems, as well as techniques to measure the effectiveness of its improvement methods. The SEI also provides access to credible information sources to help organizations successfully adopt and implement improvements. These efforts to measure and analyze processes, and to disseminate information about best practices, are the responsibility of the SEI's Software Engineering Measurement and Analysis (SEMA) group.

Our Background article provides an overview of SEMA's work, including many links where readers can obtain more information about specific improvement technologies. The Spotlight article targets one current SEMA effort: developing a practical approach to pilot-testing improvement programs.

Readers should enjoy our Roundtable on open source software. Staff members from the SEI's CERT® Coordination Center and COTS-Based Systems Initiative discuss whether open source software is a boon or bane for users and developers--and we discovered that there are some interesting disagreements on the topic. Please feel free to provide your own input on the subject via our discussion groups.

Bill Thomas
Editor-in-Chief, *SEI Interactive*

The SEI Interactive Team:

Mark Paat, communication design
Bill McSteen
Barbara White

Thanks also to Dave Zubrow, the guest editor for this issue's Features section, and to all of our content reviewers, including Steve Cross, John Goodenough, Bill Peterson, and Bill Pollak.

Introduction

Software Engineering Measurement and Analysis and Open Source Software

In this issue we depart from our past approach of focusing exclusively on one topic. This time we present articles on two subjects: Software Engineering Measurement and Analysis. We examine efforts to mature and transition measurement and analysis practices, including a new practical approach to improving pilot projects. Open Source Software. We present a wide-ranging discussion on a controversial subject that has ramifications for developers, users, and security. The following are this issue's feature articles:

Background

Software Engineering Measurement and Analysis

The SEI's Software Engineering Measurement and Analysis group works to mature and transition measurement and analysis practices. These practices enable software organizations to track process and organizational performance, leading to better control of their projects.

Spotlight

A Practical Approach to Improving Pilots

The staff of the Software Engineering Measurement and Analysis (SEMA) group have spent the past several years developing methods for measuring and collecting data from software engineering innovations. More recently, SEMA has begun to focus on ways to get in front of the process of introducing innovative methods, by looking at new ways to design pilot projects.

Roundtable

A Discussion of Open Source Software

In this Roundtable interview, we present a wide-ranging discussion of open source software—its current popularity, its advantages and disadvantages for software developers and users, the implications for computer security, and whether it is appropriate for Department of Defense systems.

Links

Links to SEIR, SEMA, and Open Source Software Resources on the WebWelcome to the eighth installment of *SEI Interactive.*

Background

Software Engineering Measurement and Analysis

The SEI's Software Engineering Measurement and Analysis group works to mature and transition measurement and analysis practices. These practices enable software organizations to track process and organizational performance, leading to better control of their projects.

Organizations that want to improve the management and control of software projects need measurement techniques and credible information sources for evaluation and benchmarking. Indeed, this ability is a requirement—either to comply with government regulations or to meet the demands of the marketplace. Federal organizations, including the Department of Defense, must comply with regulations that require the collection and reporting of quantitative performance information. With these pressures, government organizations are seeking specific guidance on how to modify their management processes to not only comply, but to improve. Within the commercial sector, competitive pressures in the marketplace provide the motivation to develop products "better, cheaper, and faster." The capability to measure and use credible and valid data is integral to survival.

Providing those measurement techniques and information sources is the responsibility of the SEI's Software Engineering Measurement and Analysis (SEMA) group.

SEMA works in three areas:

- maturing and transitioning measurement and analysis practices that enable organizations to control their projects better
- measuring the benefits and impacts of evolving SEI technologies
- disseminating information on the benefits experienced by users of improved software engineering practices

Maturing and Transitioning Measurement Practices

Current projects addressing measurement and analysis needs include developing guidance for conducting pilot projects using experimental and quasi-experimental designs (see the Spotlight article in this issue) and further investigations into the application of statistical process control (SPC) and other analytical techniques.

Building on previous work on the application of statistical process control to software engineering processes, SEMA experts are working with organizations to assist with implementation to gain additional insight into how to make SPC more effective. "Most

applications of SPC to date address the inspection process," says SEMA leader David Zubrow. "We want to better understand the practical issues of implementing SPC and to identify additional processes where this technique will yield real benefits."

SEMA transitions its work through publications, courses, and presentations. SEMA staff members have published two books to help organizations establish methods to measure the effect of introducing new technologies and processes, and evaluating how measurement methods contribute to an organization's overall business goals and objectives. The books are

- *Goal-Driven Software Measurement—A Guidebook* (Software Engineering Institute, Pittsburgh, PA, 1996)
- *Measuring the Software Process: Statistical Process Control for Software Process Improvement* (Addison-Wesley, Reading, MA,1999)

The materials in *Goal-Driven Software Measurement—A Guidebook*, by Robert E. Park, Wolfhart B. Goethert, and William A. Florac, are designed to help organizations identify, select, define, and implement software measures that can be used to support an organization's goals. In goal-driven measurement, the primary question is not "What metrics should I use?" but rather "What do I want to know or learn?" Because the answers depend on an organization's goals, no one set of measures is universally appropriate.

"Instead of attempting to develop generic, all-purpose lists of questionably useful measures, we have prepared this guidebook to describe an adaptable process that teams and individuals can use to identify and define measures that provide insights into their own management issues," the authors write. By focusing on the needs for information about software activities and using those needs to derive the measures, efforts to collect and analyze data are better able to stay focused on their intended objectives and are more likely to result in information that will be used.

In their book *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, authors William A. Florac and Anita Carleton show how well-established principles and methods for evaluating and controlling process performance can be applied in software settings to help an organization achieve its business and technical goals. Although the primary focus is on enduring issues that enable organizations to improve their long-term success and profitability, the concepts described in the book are often applicable to individual projects.

Florac says that process assessments provide a snapshot of process status at a point in time but do not answer questions about the process's consistency, effectiveness, and efficiency over time. Applying SPC to software engineering encompasses an understanding of two key concepts:

1. Process stability. Is the process behaving predictably? That is, how do engineers know whether a process is stable? Examining process performance through the use of process behavior charts allows engineers to determine whether a process is stable (within limits) and hence predictable.
2. Process capability. Is the process capable of delivering products that meet requirements and does the performance of the process meet the business needs of the organization?

Florac and Carleton recommend starting small. "Pick out key, critical processes," Carleton says. Some software organizations quickly grow frustrated with SPC because they use it to measure a big process, or even the organization's entire software process, Florac says. "They throw up their hands because the typical software process takes many months. It's best if the area of application is a sub-process of the overall development process. Pick a sub-process that is done in every development with a small number of variables, day in and day out—a test, inspection, or design process. From there, you find important process characteristics to measure and look for consistency in process performance. If you can show that the process is stable, you have a basis for improving."

Carleton and Florac have recently conducted a pilot project with NASA's Space Shuttle program, applying SPC to improve processes and gain insight into predicting outcomes and behaviors. Members of the Space Shuttle project team "do many things well, but they wanted to work with us because they wanted to improve their capability further," Carleton says. In particular, the team wants to improve its ability to predict outcomes in the software engineering process. "That is an area where these techniques are powerful and useful," she says.

To enhance the transition of these and other measurement and analysis techniques, SEMA has developed a series of courses.

Courses on Software Engineering Measurement and Analysis

The SEI has developed a measurement curriculum to help software organizations obtain precisely the data they need to advance their goals. A member of a development organization might ask, "How well is the present measurement process working for us? How often am I really surprised?" The right data will provide early warning signals, Zubrow says.

The SEI currently offers the following courses:

Implementing Goal-Driven Software Measurement
This three-day course, which can be offered as a tailored workshop for an organization, forms the basis for establishing measurement processes within organizations that

currently do not have a formal approach to software measurement. This course is most appropriate for organizations at an early level of process maturity. As a method of defining performance measures, it is appropriate for all software organizations.

Statistical Process Control for Software
This three-day course provides attendees with a practical understanding of statistical process control (SPC), and the steps associated with effectively implementing and using SPC to manage and improve software processes.

Managing Software Projects with Metrics
This three-day course teaches participants how to use software measurement to meet their goals. The course consists of lectures and exercises organized against a framework of software management activities: planning, execution and control, and review and evaluation. It provides the project manager with the tools to make decisions based upon fact rather than opinion.

Practical Software Measurement
This one-day course introduces project managers to a method for defining software measures that provides them with insight into common issues confronted by software projects.

Performance Measurement for Software Organizations
This half-day tutorial introduces managers and executives to the concepts of software measurement, common issues and challenges with respect to organizational performance measurement, and their responsibilities for defining performance measures.

SEI Public courses are offered periodically and can be attended by anyone, with a reduced charge for government personnel. In addition, customer-site courses are offered to individual organizations, and a reduced fee is charged to government organizations.

For more information about public courses and on-site training, please phone (412) 268-7702 or send email to course-info@sei.cmu.edu**.**

Working with SEI Technologies

SEMA's recent work with SEI technologies includes

- measuring the effectiveness of security practices, such as the Operationally Critical Threat, Asset, and Vulnerability Evaluation^{SM} (OCTAVE^{SM}) method for identifying and managing information security risks (see the SEI's publication on OCTAVE)

- measuring the impact of product line practices (see our September 1999 feature on product line practices, as well as the SEI Web site at http://www.sei.cmu.edu/plp/plp_init.html)
- producing validation reports on the Architecture Tradeoff Analysis Method℠ (ATAM℠) (see our column The Architect, as well as the SEI Web site at http://www.sei.cmu.edu/ata/ata_init.html)
- assisting in the development and validation of cost-estimation practices and models for the integration of commercial off-the-shelf (COTS) software (see our June 1998 feature on COTS and our column The COTS Spot, as well as the SEI Web site at http://www.sei.cmu.edu/cbs/index.html)
- piloting and demonstrating the benefits of model-based verification as a tool to improve the software development process in a project with Hill Air Force Base. In this application of model-based verification, requirements and specifications for a real, already-deployed, software system are being modeled with the expectation of being able to identify defects earlier in the life cycle, which generally results in lower costs for correcting those defects. (See the SEI technical reports *Model-Based Verification: A Technology for Dependable Upgrade* and *A Study of Practice Issues in Model-Based Verification Using the Symbolic Model Verifier*.)
- assisting with the implementation of the test and evaluation master plan for the Capability Maturity Model® Integration (CMMI℠) project (see our December 1999 feature on CMMI and the SEI Web site at http://www.sei.cmu.edu/cmm/cmms/cmms.integration.html
- performing data collection and analysis to assist with the development of the Team Software Process℠ (TSP℠) (see our June 1999 feature on TSP and our column Watts New?, as well as the SEI Web site at http://www.sei.cmu.edu/tsp/)

Disseminating Information

Many organizations seek information to help them decide whether to adopt new technologies and practices or to embark on process improvement. The Software Engineering Information Repository (SEIR) fills this need by providing a central source of information on the implementation of software engineering improvement methods and practices.

SEIR users typically seek information they can use to gain further sponsorship from their organization to continue their improvement efforts. Users can read about other organizations' experiences, lessons learned, success stories, and benefits of implementing improvement methods and practices. In addition, the SEIR accepts work aids and artifacts contributed by its users. These often serve as a starting point for many organizations looking for ideas about how to begin implementing a new practice. "This type of information is most helpful when it comes from the users themselves," Zubrow says. Major companies, more than 60 colleges and universities, and many DoD organizations are represented among more than 6,000 SEIR users and contributors.

Additionally, SEMA is revitalizing the Software Technology Review, a collection of short descriptions on more than 60 software technologies. It gives a description and objective assessment of current technologies in a format that is concise and easy to read. The original intent of the *Software Technology Review* was to help managers and executives keep pace with the ever-increasing variety of technologies and innovations. However, the review is also useful to engineers who want to quickly scan what's new.

If you are interested in the work of SEMA, please visit the SEMA Web site for more information about current projects and opportunities for collaboration.

Spotlight

A Practical Approach to Improving Pilots

The staff of the Software Engineering Measurement and Analysis (SEMA) initiative have spent the past several years developing methods for measuring and collecting data from software engineering innovations. More recently, SEMA has begun to focus on ways to get in front of the process of introducing innovative methods, by looking at new ways to design pilot projects.

SEMA leader David Zubrow and SEI technical staff member Will Hayes point out that pilot projects have long been regarded as a smart way to gain experience with a new idea. "But practical limitations often erode the good intentions of the professionals who conduct pilot studies," Hayes says. "Immediate customer needs are more important than research. Political influences—pro and con—can come into play. And reporting a negative result could be unacceptable."

Also, a pilot study that is viewed only as a feasibility analysis leads to a go/no-go decision, Hayes says. "The business context is rarely that simplistic. Utility, not feasibility, is what matters. The conditions that influence utility must be identified, as well as options and mitigators. A binary outcome is too limiting."

SEMA's goal is to develop guidance for designing and conducting effective and efficient pilots. Zubrow says pilots will be effective if they yield results that support good decisions, and efficient if they consume as few resources as possible to achieve the desired level of confidence. The guidance for such pilot projects should address the design of the project and its measurement methods, the data collection, and the analysis of that data.

"The problem lies in the fact that conducting pilot projects and evaluations is a common practice, but little guidance exists," Zubrow says. "An opportunity is missed to leverage the results and experiences of previous pilots. There is also a significant risk in applying the results of a pilot to the broader organization if the pilot was not designed properly."

The question, Zubrow says, is "how do you design the pilot in a way that gives you the best information, given that there is a set of constraints: time, disruption, cost, et cetera."

Hayes and Zubrow have proposed the use of quasi-experimental techniques and meta-analysis to provide powerful, low-cost, methods for going beyond the current practice for conducting pilot studies.

Quasi-Experimental Design

The "quasi-experimental" approach is so called because it does not meet the standard of a true scientific experiment. But such an approach is often more feasible given the constraints of time, cost, and allowable levels of disruption in an organization.

A pilot that uses a quasi-experimental design strives to approximate, in a field situation, a true experimental design, which requires the random assignment of subjects, control over their exposure to experimental treatment, control over other influences, and a large number of observations. In quasi-experimental design, allowances are made for the fact that it may be impossible to fully conform to these requirements. Zubrow points to three types of quasi-experimental designs:

- *time series*, which involves periodic measurement with the introduction of an innovation
- *non-equivalent control group*, in which the innovation is given to one candidate and another is selected to be a control
- *multiple time series design*, which is a combination of the time series and non-equivalent control group designs

In a time series design, repeated measurements of performance are made prior to and after the introduction of the improvement. This approach requires the implementation of only one pilot project, but lacks history and is difficult to use for generalizations. It works best when observations and measurements are taken for an extended time before and after the introduction of the improvement.

In a non-equivalent control group design, the innovation is applied to naturally occurring groups, rather than through random assignment. Measurements are made at comparable times both prior to the introduction of the innovation and after it. There is no assurance that the groups are equivalent prior to the "experiment." In this method, shorter observation times are required, but at least two groups are needed and the lack of equivalence among the groups is a weakness. This method is useful when there are limited opportunities to gather data over time and when there are two comparable situations in which the innovation can be tried.

The preferred approach, when possible, is to use a combination of the time series and non-equivalent control group designs—the multiple time series design. It provides improved confidence in the results, but requires increased data collection and coordination.

Meta-Analysis

Meta-analysis has been defined as "the statistical analysis of a large collection of analysis results from individual studies for the purpose of integrating the findings.[1]

With meta-analysis, study results and testimonials can be treated as data, with no requirement to accept one result and reject others. A weighting scheme can be devised based on such objective criteria as the amount of background information provided, the similarity of the context to the project at hand, the timeframe of the study, and the subject pool involved. The data can be employed to construct prediction intervals for the pilot study, and to provide added context to interpret pilot study results.

The use of research has limitations, Hayes points out, because typically only positive results are published. Also, use of student subjects in academic settings leads to skepticism about the applicability of the results to the "real world." Extrapolation beyond the research setting is typically an act of faith, not driven by quantitative methods.

Combining published research findings with pilot results can help minimize the weaknesses of both sources of information:

- Pilot study results can be estimated using published results from experiments.
- Multiple studies can be used even if they report conflicting results.
- Important differences in study design or subjects used can be incorporated into the analysis.
- Multiple pilot studies can be analyzed in combination.

Designing pilots with meta-analysis in mind can further add value. "We want to design pilots to make the data amenable to meta-analysis. That way the organization builds a body of knowledge about itself and its capability to improve," Zubrow says.

References

Campbell, D.T. and Stanley, J.C. 1963. Experimental and Quasi-Experimental Designs for Research. Boston: Houghton Mifflin.

Hedges, L. V., and Olkin, I. 1985. Statistical Methods for Meta-Analysis. Orlando, FL: Academic Press.

---

[1] Glass, G.V. "Primary, Secondary, and Meta-Analysis of Research." Educational Researcher, Vol. 5, No. 10. pp 3-8. 1976.

Wolf, F. M. 1988. Meta-Analysis: Quantitative Methods for Research Synthesis. Beverly Hills, CA: Sage Publications.

Roundtable

A Discussion of Open Source Software

Moderated by Bill Thomas

In this Roundtable interview, we present a wide-ranging discussion of open source software—its current popularity, its advantages and disadvantages for software developers and users, the implications for computer security, and whether it is appropriate for Department of Defense (DoD) systems.

The participants are Chuck Weinstock, Pat Place, Scott Hissam, and Dan Plakosh, all members of the SEI's COTS-Based Systems Initiative, and Jed Pickel of the CERT® Coordination Center at the SEI. The moderator is Bill Thomas, editor of *SEI Interactive*.

The topics covered are:

Why Do Research on Open Source Software?
Is There a Community of Developers?
Who Has Accountability for Open Source Software?
What Are the Benefits and Drawbacks of Developing Open Source?
Is Open Source Right for the Department of Defense?
What Are the Criteria for Success in Open Source Development?
What Are the Advantages to the User?
What Are the Security Implications of Using Open Source Software?
What Comparisons Can Be Made Between Open Source and COTS?

Why Do Research on Open Source Software?

Bill Thomas: Why do you think that open source software warrants research at this time?

Chuck Weinstock: One reason is that it appears that the community is treating open source software as the next silver bullet. We all know that silver bullets very seldom find their target, and the community moves on to the next silver bullet.

Pat Place: I would add that there is a substantial amount of software available these days that is open source. If you are interested in building systems out of existing components—be they open source or any other form of source—you need to understand at least the risks as well as the benefits of doing so. If we can say anything that helps, then I think that's a good thing.

Scott Hissam: The phenomenon that is happening with the Linux environment is getting everybody's attention to open source software—more attention than has ever been paid to it before, at least in the media. People are enamored and believe that Linux is a successful, stable development environment, and that somehow every piece of open source software that they get is going to be just as stable and just as reliable as the Linux platform—if you believe that it is as stable and reliable as it is touted to be in the press.

Jed Pickel: Open source has been around for a long time—probably more than 20 years. I think one of the reasons why it's getting so much attention now is because commercial interests are developing it. That's why we're seeing the media interest.

Is There a Community of Developers?

Dan Plakosh: You can release your source code, but I'm not sure people really know what to do with it. I released open source software two years ago and I've had very few people dive into developing it.

Place: It gets even worse when you get something that is hundreds of thousands of lines or a million lines. The historical aspect is interesting because you can look back at the history of programs that were open source, or close to open source, with lots of people helping and providing fixes. You can start to see what was successful about them and what is different about what is becoming the current open source movement, which I honestly believe is going to lead to disaster. I can provide anecdotal evidence of examples where you've got something like a `tcsh` [an expanded version of the original C shell for the UNIX operating system] which is not that complicated a program, but has features and peculiarities that are so weird that you'd never even want them. And yet somebody has said, "Oh, I'll just go and stick this thing into the system." For example, in `tcsh`, if you have time displayed as part of your prompt and it happens to hit the hour, it'll go "ding" instead of printing the time. I mean, this is insanity: feature upon feature upon feature that leads to code that's got more junk in it than you can possibly be interested in. It ends up becoming ultimately unmaintainable code.

Hissam: But I would say that the `tcsh` example that you gave is an unbounded development activity that nobody really paid attention to. Nobody really cared about it and that's why it got unwieldy. Not every piece of open source software is developed in that way.

Place: That's exactly true. I think that's the key to the difference between those things that have been and will be successful and those things that will not be successful. Somebody or some very small group of people have a very clear idea as to what that system is going to be, what it's going to do, and how it's going to be architected. And they keep it that way.

Weinstock: Some people refer to those people as the "arbiters of good taste."

Place: That's the phrase that was used primarily about the original UNIX developers. They were arbiters of good taste. With all of the stuff that people from all the universities shipped to them back in the mid-1970s and early ’80s, they decided what went into the source and what was not in the source. For the longest time with Linux, Linus Torvalds [the Finnish graduate student who created the original Linux operating system] was the person who did that. He had a vision as to what it was going to be. That seems to be drifting out. Linux is perhaps losing some of that arbiter-of-good-taste quality.

Pickel: On your `tcsh` example: there are plenty of examples of closed source software having very similar things. For example, one commercial product is such that if you hit certain key sequences, you can end up with a flight simulator—which is a little bit different from `tcsh` beeping at the end of a line. The difference, though, is that should the community come across that item in `tcsh`, and feel like it needs to be removed, and there are enough people who agree with that, then it would be removed from `tcsh`. You can easily go and change that if it bothered you enough.

Place: Absolutely. I've done that because I wanted `tcsh` to be as small as possible and I’ve used it as small as possible.

Hissam: So you removed a whole bunch of things out of `tcsh` that you didn't like. Right? Now let’s say the next version of `tcsh` comes out and you want to adopt it.

Place: I have developed the version of the shell that has the capabilities I need. If anything does come out in `tcsh` that I'm interested in, I might take that as a patch file and patch my source with those changes. But I'm not taking all their stuff again.

Weinstock: You now own the problem.

Place: Yes, absolutely. I willingly accept that. Of course, the advantage is that it was open source. I could choose to take on the risk and build something that was what I wanted.

Who Has Accountability for Open Source Software?

Thomas: It seems that no one has any accountability with open source software. It’s strictly “buyer beware.”

Pickel: I would disagree with that. Let's go back to the `tcsh` example again, because I think that's a good one in that a person maintains it and is accountable for listening to the users. Pat didn't speak up in this case. He decided to split off his own version and now he's accountable for that.

Place: If you look at the actual source code, you'll see all of these different names of people who've added this and added that. The risk I see with open source is that all of these features are getting thrown into a basically good system. Someone wants the X widget or the Y widget, so they just go and put that fix in and you get this loss of a sense of stability of the project—this loss of sanity. `tcsh` going "ding" is kind of stupid. It goes to my notion of good taste; it's below the cut line.

Pickel: What you're describing is an example of open source working in the most optimal sense in that there are people who have different goals from a project than you do. You decide to split off your own version; that's open source working right there.

Hissam: It depends on what your own goals are. If your own goals are to keep up with the latest and greatest, then him vectoring off his own version—he's stuck.

Place: That's disaster if you want to keep up with the latest.

Plakosh: I don't think people develop open source software with the intent that people will take it and go off and build their own products from it. I think it's more so that people will contribute to mature whatever piece of software that they're doing.

Place: The freedom for anyone to make a change leads to the fact that the product will never mature because it will always be in a state of flux.

Plakosh: There is not the freedom for anybody to make a change. I really think you're looking at isolated cases. For example, take Linux. The majority of people who use Linux never look at the source. In the majority of most open source products, I would bet that people do not look at the source. They don't care. They don't recompile it. They don't want to have anything to do with it. It's only the people who are working toward the development of Linux who are looking at the source, or occasionally someone who has the in-depth knowledge finds a bug and looks at the source. They may fix it but they may also submit it to one of the Linux working groups to have it corrected.

Hissam: Let’s go back to the earlier question about accountability. We disagreed about whether anyone is accountable. What does it mean to be accountable? It means that there's liability on the part of somebody.

Place: I wouldn't even go as far as that. Dan has released source. He's put his name up saying, "I think this is a good piece of source." In an open source project, other than a couple of special cases, there's a substantial body of existing code that gets released and then people can work on it, rather than working on stuff from scratch. I see a potential split there. Take Linux. Who is accountable for Linux these days? Does Linus put his name on it saying, "I think this is all good source code"? I don't think so anymore.

Hissam: No. The worst thing that can happen to the people who are "accountable" for Linux is that the world turns their back on it. But concerning accountability: I think the answer is that *no one* is accountable outright and I think it is buyer beware.

Weinstock: So that's why people go to places like [Linux vendor] Red Hat instead of just downloading it off the Web.

Hissam: Because they want to hand money to somebody. They want to be able to say, "Give me this and give me that."

Weinstock: Red Hat also sells support. You can go just buy a Red Hat CD and you get nothing with it other than the CD. But you can also go to them and get support for Linux.

Pickel: That's how Linux has made it into the corporate world: doing support.

Plakosh: I've dealt with support before and support is not typically all that it's cracked up to be. Support is usually geared toward people who have problems in reading the documentation or who don't understand things. Linux got into the commercial domain mainly due to the attractiveness of it being free and being somewhat reliable.

What Are the Benefits and Drawbacks of Developing Open Source?

Thomas: Let's backtrack a little bit here. What would you say are the benefits and drawbacks of developing software in an open source environment, from the standpoint of the developer?

Weinstock: There are different ways of looking at that. Why would I want to participate in the development or why would I want to put myself more out there for free...

Place: I'll tell you at least one person's motivation for the latter. For the last two years, he's been unable to further his software at all, so since it was previously freely available, he said, "Okay, let's make this an open source project in the official open source project way. We'll get people who have ideas and have some suggestions for developing this further, and/or who have bug fixes to be able to maintain this thing."

Hissam: So, would you say that the rate of change on this project has increased or decreased, and have those changes been dramatic?

Place: Well, it's certainly increased. There's also one place where you can get an official source version that has the bug fixes in it, which you couldn't do previously.

Thomas: Would you say that putting out a program with open source code is a way of testing the market for it?

Pickel: Exactly. That's another good point that I wanted to make. One of the interesting things about open source is that you build on other people's software. When you release something, you never quite know how other people are going to make use of it. You learn quickly that way because they give you immediate feedback and contribute changes. It's a great way to figure out market demand.

Hissam: That would be a benefit. But if that evolution is unchecked, you're going to get the `tcsh` phenomenon. It's almost like a cancer: cancerous features.

Pickel: You choose the branch of the code that most suits your goals at a given time.

Weinstock: But that presents the consumer with a real problem, right? Which branch? What happens to the uneducated consumer who doesn't have a basis for picking a branch?

Pickel: They go to places like Red Hat.

Place: If you want a version of BSD [a popular version of UNIX; BSD stands for Berkeley Software Distribution], which one do you pick right now? There are three versions of BSD that are all based upon 4.4, BSD Light, which was the last release. So which one do you choose?

Weinstock: Getting back to what you said about consumers going to Red Hat because they don't know how to make that choice: That's fine for an open source project where there is a Red Hat, but my guess is that most of them don't have a Red Hat. I mean, how do I know which Emacs to choose, for instance?

Plakosh: The only reason you have companies like Red Hat out there is because the distribution package for Linux is so large and so complicated—or at least it's viewed that way by the consumer. For a small piece of software, you're not going to have these distributors.

Pickel: Going back to your point about what to do if there is no Red Hat: I think these companies are out there for the purpose of infiltrating the corporate world—getting this kind of software into the corporate world. The techies and the geeks aren't necessarily interested in a Red Hat, though they may use it because they don't necessarily care to package all the software. But there are projects that don't have corporate backing behind them, or a very organized way of going about things. They're just not going to make it to as large an audience. They won't make it into the corporate world quite as easily.

Hissam: Every techie and geek on the planet right now, and every open source activity, started with dreams of IPOs [initial public offerings of stock]. They want to be the next millionaire. They want to start the next company.

Pickel: If you look at the past year or so, that might be one of the motivations behind open source software: people think they're going to make a killing off it. If you look over the past 20 years, there haven't necessarily been financial reasons. One of the ways you get paid for leading a successful open source project is by getting your name out there, by getting well known, by becoming the guy who started that project.

Is Open Source Right for the Department of Defense?

Weinstock: Let's talk about open source as applied to our Department of Defense client. What are the advantages of developing something using the open source model? When applied to the DoD, the notoriety factor is probably not important to them.

Place: There's another question that I should like to raise with respect to DoD customers. What systems do you envision the DoD would like to build with open source? Is it the weapons systems? Is it the payroll systems? How many people out there are interested in the payroll system?

Weinstock: It would seem to me that we're probably talking about subsystems first of all. Pieces of systems.

Place: So we're talking back at the level of things like the operating system (OS), the database, the underlying components—the bits that we take for granted. That's one of the issues, when we talk about DoD customers. We need to understand that they're not going to build systems with this stuff.

Weinstock: But they will build systems that contain parts.

Place: Then the question is, which parts? Clearly it's going to be exactly those things—the OS parts, the database parts, the GUI [graphical user interface] parts.

What Are the Criteria for Success in Open Source Development?

Hissam: We can go back to the premise that these large organizations, be they DoD or not, think that they can get something done with access to a large, talented pool of engineers. There's some belief that they can get access to a lot of peer reviews, beta testers, people out there to look at their software and make it better and get it done quicker. That seems

to be the running belief. If that's a model of success in Linux, then it must be true for every piece of open source software. But we should debunk that. Past performance should not be used as an indicator of future performance. That's one of the reasons that the Software Engineering Institute has to start looking at the processes that are used in open source development. What are the criteria that have to be there in order for it to be successful?

Place: There are instances of projects that have been very successful. I would claim that Linux certainly has been one of them. It has achieved a level of reliability. The BSDs are reasonably successful, and some other open source things are reasonably successful. One of the common themes I've seen through either open or freely available source projects over the last 20 years is that there has been a substantial body of software—i.e., the system is basically there—before its release. People are bug fixing rather than developing new features, so that you've got a system with a structure and a design, and other people are now fixing the things that "he forgot" or that “he got wrong.”

Weinstock: That suggests that it should start off with a user base or some sort of base of people who care about it.

Place: You certainly need people to care about it, and the people who care about this typically are the users.

Plakosh: But if you look at how Linux kicked off, it kicked off by being more or less the toy of software engineers to build upon. It came with a lot of people's research projects. There were a lot of people looking at this functionality and that functionality. The user base of Linux was people who developed software, not users of software, per se.

Place: That’s a good point. The other thing, in thinking about what has been successful, is that the initial release was something that was a well-designed system.

What Are the Advantages to the User?

Thomas: Let's talk a little bit about open source from the user's perspective. What are the advantages to using open source software?

Hissam: Let me start off by cutting to the chase. It's a two-edged sword. The advantages are that users can get the latest and greatest and the fastest fixes. The disadvantages are that they *have to get* the latest and greatest and the fastest fixes. They might spend 75% of their time using a product and 25% of their time upgrading the product.

Plakosh: That's not necessarily true. Just because a product is out in open source doesn't mean that I, as a user, have to track it. There are a lot of internal releases that I don't need

to worry about or that I may not want to worry about. That perspective is somewhat from the mentality of the world that we live in, developing software.

What Are the Security Implications of Using Open Source Software?

Pickel: From a security perspective, you could look at it from a couple of standpoints. You really have to pay close attention to the software because if the community becomes aware of a vulnerability, then they're going to exploit it. So you need to beat them.

When you were talking about the double-edged sword, I realized that it also applies to the perspective of the developers in that there's an advantage to having people working on your software, but you also have to be ready to deal with them. If you have a lot of demand, and a lot of people developing your software, it's going to be tough to deal with them.

Place: You raised the issue of security. What trust do you place in open source software, given that it's changing so rapidly? How much analysis can you do on the 5,000 fixes that came in last week?

Weinstock: Do you use Linux in a secure environment?

Pickel: Absolutely. Actually, I run Linux on all my machines. I'm not going to look at every single line of code. I'm not going to look at every update. But the thing is that there are people out there who are.

Weinstock: You hope. Do you believe that every nook and cranny of Linux has been looked at with that in mind?

Pickel: Not necessarily. But I believe that a lot more nooks and crannies have been looked at than are looked at in closed source environments.

Plakosh: That's an interesting statement because I would tend to bet that there is a difference between theory and practice. In theory, you would think that you're releasing source and you would have people combing over the code looking for security holes and trying to fix them. In practice, I don't think that's necessarily the case. In theory, it sounds great: the more people have it, the more people are looking at the source code and the more people are going to try to find bugs or security holes so that they can try to fix them. That sounds great. In practice, I think the only people who are trying to do that are maybe people who are trying to break into a system.

Pickel: Exactly. And they're part of the community. If they identify a hole and start exploiting it, people will notice that.

Hissam: You're saying that even the bad guys, in a sense...

Pickel: ...they're part of your development.

Place: But you only find out after the fact.

Pickel: That's still a better environment than closed source.

Plakosh: Actually, that's *not* necessarily much better because some of these holes that you can find in open source code you never would have found if it was closed source. They never would have existed.

Hissam: If I were a hacker, I could get the latest distribution from Red Hat, go to my garage, close the doors, get a lot of Twinkies and Coke, and start mulling over it until I can find a hack. Then I can just turn on my modem and go attack somebody who's using Linux.

Pickel: Certainly, there's some potential of administrators not noticing. This is an issue that we deal with every day in the CERT® Coordination Center. It's quite common that somebody will break into a site and the site administrators don't know how it happened. But when you deal with the sophisticated administrator, you can usually track down what program was the source, the problem. Then, maybe there's a new vulnerability in that. So, a lot of times, we look through the vulnerabilities, get in contact with vendors, and have the problem fixed. So there, in that situation, a few people were compromised. But the community as a whole is now operating on more secure software.

Place: Then there is the difficulty of getting customers to upgrade with the patch. Don't underestimate the number of people who are behind the curve.

Thomas: Is it any better in a closed source situation?

Hissam: Closed or open source, when a vulnerability is discovered, people have to react. I think the other thing that is interesting is that a hacker can become very intimate with some of the underlying protocols that are used. There may be that somewhere in the code it says, "You'd better check for null value here and the packet header for this because if you don't you could lock up the machine." The hacker says, "Wow! I hadn't thought of that before. If I tried this against Windows, I wonder what it would do?" Just by reading the source code, they could learn some very sophisticated, obscure attack.

What Comparisons Can Be Made Between Open Source and COTS?

Weinstock: There's also a big push to use COTS [commercial off-the-shelf] software, and widely using COTS raises all sorts of problems. At least some of the same arguments that apply to COTS apply to open source software.

Place: You might get an instability argument more so with open source than with COTS.

Plakosh: You'll get quality arguments too.

Weinstock: But it's the argument that "you own the solution if you try to modify it in any way."

Hissam: Let's say a contractor is using open source in a government program and they're having some success. Then they run into a roadblock. They decide, "All I have to do is change this one line of code and we'll save the government millions of dollars." And they do it. Now let's say the open source community doesn't want to adopt that change because it's very specific to whatever the government is doing. Now the government—by virtue of a contractor—is in the business of maintaining and competing with that open source.

Plakosh: That may or may not be true. I think one of the best advantages to me as a software developer is to take somebody else's open source and save development time, if it does do that for me. And I don't intend to track it in the future.

Weinstock: Yes, but suppose there's a security compromise that's found in some future version and the four-star says, "That rolls back to the version you modified seven years ago—or six months ago." Now you've got to find someone who's even smart enough to put the changes in.

Plakosh: No you don't. You've taken over maintenance of that. That's fine, and you move on.

Weinstock: And the technician who worked on it seven years ago is still there, on the payroll, and there ready to help?

Plakosh: No. But we're just talking about another method of software reuse. You're equating it to a product and having to track versions, rather than someone looking at open source software and saying, "Man, I could use these features. Let me rip them out and use them." You've taken the code from an open source product and reused it elsewhere.

Weinstock: But you've lost a supposed benefit of open source, which is that vast community of developers.

Plakosh: But if you weren't interested in that, it doesn't matter. You've gained. You didn't have to write that. You didn't lose anything. That's the benefit to a lot of people who are using open source. They don't have to reinvent the wheel. Somebody's already invented it. Yeah, I have to maintain it. Yeah, I'd better take a look at what I'm getting. Yeah, I'd better check the quality of it.

Weinstock: I'm not disagreeing with you. That's certainly a valid, good thing about open source. But if the whole world had that view of open source, there wouldn't be open source. All I'm saying is that it's sort of outside the spirit of open source as I see it.

Plakosh: What's the spirit of open source? Open source has two motives. One is that you want other people to work on your code; you want resources. So you want to obtain free resources.

Weinstock: Right. And you taking the code and going you own way and not feeding back to the community does not accomplish that.

Plakosh: But that's one motive. It's for your own—how can I put it—personal glory, corporate gain, whatever. The second advantage is for other people just to advance technology and to advance people's growth. People give things out so that some other people can learn how to do something. For other people, it fosters new ideas. That's why I give out source code. I don't do it for my personal gain. I do it so that other people can look at it and use it for whatever they want to use it for.

Weinstock: That's from a developer's perspective.

Plakosh: I think it's great if someone reuses and finds benefit from something that I wrote, and if it saves them some time. Just like if I'm going to write a piece of software, I go out looking. I'm not going to try writing everything from scratch when I know that there is software that I can lift.

Pickel: The real benefit of open source, in my opinion, is that you build on things that are already available. You're furthering technology, and then people build on what you've done.

Place: You get the function you want as well. That's the other thing. If you're buying from Chuck's House of Software, you get what Chuck wants to sell you, not what you want.

Pickel: But you do have to be a developer to get that. I've been a user/developer of these things for a long time. If you're not a developer, you don't get it. I suspect that one of the results of this is that there will be more people who are developers out there. It's going to convince more people to look at the source code.

About the Participants

Scott Hissam is a member of the technical staff in the COTS-Based Systems Initiative at the SEI. He has 14 years of software development experience in industry, defense, and research. Hissam's principal areas of expertise include distributed applications, secure software and systems engineering, networking protocols, and operating system internals. Much of his recent experience has been rehosting legacy database applications on the World Wide Web. Prior to the SEI, Hissam was with Lockheed Martin, Bell Atlantic, and the U.S. Department of Defense.

Jed Pickel is a member of the technical staff at the CERT® Coordination Center (CERT/CC) at the SEI. Pickel is a member of the CERT/CC Incident Response Team. Pickel earned a Bachelor of Science in Electrical Engineering from the University of California at San Diego and is pursuing an MS from the Information Networking Institute at Carnegie Mellon University.

Patrick Place is a member of the technical staff in the COTS-Based Systems Initiative at the SEI. Recently he has been part of the development of COTS Usage Risk Evaluation (CURE), a method for early identification of COTS-based risk in system acquisitions. Prior work at the SEI has included investigations into communications in distributed real-time systems as well as system specification using formal techniques. He has also developed tools to support Web development at the SEI. Before joining the SEI, he worked at various companies either developing tools or formal specifications, sometimes both. He has been an adjunct lecturer at South Bank University in the Electrical Engineering Department and also at Imperial College in the Computer Science Department.

Dan Plakosh is a member of the technical staff at the SEI. He is investigating technologies, techniques, and methodologies for developing distributed systems using commercially available distributed object technologies. He is currently researching topics associated with component architectures such as the development of custom component frameworks. Prior to joining the SEI, Plakosh was the lead software engineer for the Systems Engineering Department (F31) at the Naval Surface Warfare Center. Plakosh has 14 years of software development experience in defense, research, and industry. Plakosh's principal areas of expertise include real-time distributed systems, network communications and protocols, systems engineering, real-time 2D and 3D graphics, and Unix OS internals. Much of Plakosh's recent experience has been redesigning legacy-distributed systems to use the latest distributed communication technologies.

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.

Chuck Weinstock is a member of the technical staff in the COTS-Based Systems and Dependable Systems Upgrade Initiatives at the SEI. Weinstock is currently involved with helping the U.S. Coast Guard to understand COTS-related issues in the procurement of a new fleet of cutters. He is also working with Hill Air Force Base in a study of the use of model-based verification techniques in the software development process. He and Scott Hissam are embarking on an effort to better understand the open-source phenomenon. Before coming to the SEI, Weinstock was on the staff of Tartan where he worked on optimizing compiler technology. Before that, he was with SRI International's Computer Science Laboratory where he worked on the Software-Implemented Fault-Tolerant (SIFT) computer system.

Links

Resources on the Web

Software Engineering Measurement and Analysis

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

The mission of the Software Engineering Measurement and Analysis (SEMA) group is to help software organizations who want to enhance their capability to improve and manage software projects through the use of data-driven decision making. Visit this site for information about related products and services and information resources.

Software Engineering Information Repository

http://seir.sei.cmu.edu

The Software Engineering Information Repository (SEIR) provides a forum for the contribution and exchange of information concerning software engineering improvement activities. Registered customers of the SEIR can exchange questions or tips and contribute (deposit) experiences or examples to assist each other with their implementation efforts. Visit this site for additional information and news about how to register to participate in the SEIR.

Open Source Software

http://www.opensource.org

This site offers several complementary views of open source software. At this site, you'll find a brief introduction and third-party case studies, as well as case studies from hackers, executives, and customers.

http://www.openbsd.org

OpenBSD project members produce a free, multi-platform 4.4BSD-based UNIX-like operating system. Visit the site to learn more about the goals of the project and how the effort is funded.

http://www.freebsd.org

FreeBSD is an advanced BSD UNIX operating system for PC-compatible computers.

FreeBSD offers advanced networking, performance, security, and compatibility features. Visit the site to learn more about the project, download FreeBSD, or access the documentation.

http://www.netbsd.org

The NetBSD Project is an international collaborative effort to produce NetBSD, a freely available and redistributable UNIX-like operating system. In addition to this effort, NetBSD contains a variety of other free software, including 4.4BSD Lite from the University of California. Visit the site to read the latest news releases, or to get more information about distribution and support.

http://www.gnu.org

The GNU Project was launched in 1984 to develop the GNU system, a complete Unix-like operating system. This site contains information about the software and how to download it, and the project's history and future plans.

http://www.freshmeat.net

This site keeps track of current versions of software used on Linux systems that are shipped to customers.

http://www.sourceforge.com

SourceForge's mission is to enrich the Open Source community by providing a centralized place for Open Source Developers to control and manage Open Source Software Development. Visit the site for information about software, new releases, and documentation.

http://www.linux.com

This site provides a centralized place for individuals of all experience levels to learn about the Linux Operating System. Visit the site for Linux-related chat rooms, games, hardware, software development, security, etc.

http://www.opensales.org

OpenSales.org is the development site for OpenMerchant™, an Open Source e-commerce Internet application written in Perl. At this site, you'll find resources and documentation to run OpenMerchant, and you'll have access to the OpenMerchant source and the development community.

http://www.collab.net
This site provides information about how to bring open source and business together using internet-based collaboration.

http://www.mozilla.org
Mozilla is an open-source Web browser, designed for standards compliance, performance, and portability. At this site, you'll find discussion forums, software engineering tools, information about releases, and tools for bug tracking.

http://lifelines.sourceforge.net
LifeLines is a genealogy program to help with your family history research. Its primarily strengths are a powerful scripting language and the ability to import and export information easily in the GEDCOM format. Find related information and a history of this project at the site.

http://www.gartnerweb.com/public/static/hotc/hc00085832.html
This site a reality check on the origins of--and the current players associated with--open source software.

The Architect

An Architectural Approach to Software Cost Modeling

Jai Asundi, Rick Kazman, Mark Klein

The aim of a successful software project is to maximize the difference between the value of the software product and its cost. The value of a complex software-intensive system results from the interaction of the functionality and quality of the software, and the marketplace. Traditional cost-estimation models [Londeix 87] have concentrated on methods to estimate the "as-built" costs of simply building the software products. Some cost-estimation techniques at the design stages [Jones 98] only consider the costs of functionality through the use of function points. However, the costs incurred for meeting the system's quality goals are never estimated. These techniques also do not consider the costs that occur during the entire lifetime of the product, including stochastic events such as the failure or change of components, change of platforms, or even change of methods of communication among components.

A software architecture is a critical part of complex software-intensive system design. As Shaw states, "in a situation of *increasing size and complexity, the design and specification of overall system structure becomes more significant"* (emphasis added) [Shaw 96]. The term "overall system structure" refers to the software architecture. Software architects usually do not have well-developed structures to reason about the costs of design options. The Architecture Tradeoff Analysis Method (ATAM) [Kazman 99] provides a software architect with a framework for reasoning about the technical tradeoffs. the attribute-based architectural styles (ABAS) framework [Klein 99] used within the ATAM helps an architect reason, both quantitatively and qualitatively, about the quality attributes and the stimulus/response characteristics of the system. For example, ABASs allow one to reason about important system stimuli (the failure of a component, a user request, a change to the software) and the associated system responses (mean time to failure, latency, or person-months of labor, respectively).

Functionality is key in satisfying marketplace needs, but functionality is worthless if it is not endowed with the right quality attributes; and it is these qualities that shape the architecture and, consequently, dictate cost. We propose that an architectural approach to cost estimation is necessary to enable reasoning about a system's long-term costs as well as the coupling of costs and technical tradeoffs in the architectural design.

Problem Framework

Our understanding of the cost-estimation problem arises from the idea that any software project is the result of a set of business goals that emerge from a desire to exploit a niche in the marketplace with a new software product. Take, for example, the development of an application server that caters to on-demand software. The business goals of having a robust, high-performance, secure server lead to a set of architectural decisions whose goal is to realize specific quality-attribute requirements of the system (e.g., using tri-modular redundancy to satisfy the availability requirements, a dynamic load-balancing mechanism to meet the performance requirements, and a 256-bit encryption scheme to satisfy the security requirements). Each architecture A that results from a set $\{A_i\}$ of architectural decisions has a different set of costs $C\{A_i\}$(Fig. 1). The choice of a particular set of architectural decisions maps to system qualities that can be described in terms of a particular set of stimulus/response characteristics of the system $\{Q_i\}$, i.e., $A_i \rightarrow Q_i$. (For example, the choice of using concurrent pipelines for servicing requests in this system leads to a predicted worst-case latency of 500 ms, given a specific rate of server requests.) The "value" of any particular stimulus/response characteristic chosen is the revenue that could be earned by the product in the marketplace owing to that characteristic. We believe that the software architect should attempt to maximize the difference between the value generated by the product and its cost**.**

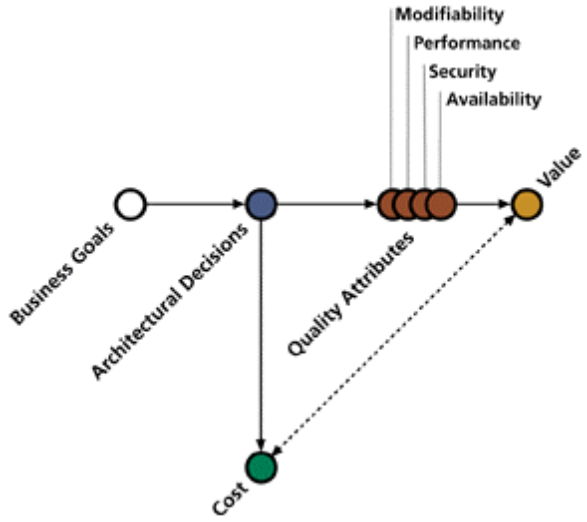

Figure 1: Business goals drive the architectural decisions $\{A_i\}$, which determine the quality attributes $\{Q_i\}$.Value(V_a) depends on Q_i and Cost(C) depends on A_i.

Developing a Cost Estimate

The cost of a system can be broken up into cost of the initial design, cost of implementation, maintenance and evolution of the system, and the future costs[1]. Design activity could be of two kinds: routine and innovative. For routine systems, cost estimation is not a very difficult task because the architects/designers know a lot of details about previously built systems and have a good handle on costs. Traditional cost-estimation techniques are very good for these kinds of systems and can easily provide estimates of the cost of the implementation, maintenance, and evolution of the system. However, thinking about the probable future costs—such as complete systemic changes or even costs arising due to stochastic events like the loss of support for a commercial-off-the-shelf (COTS) component—is more challenging. On one hand, what time scale should one choose to ascertain probable future costs? Given that the costs should be represented by a probability distribution, how does one obtain those numbers to do quantitative analysis? We believe that the choice of time scale and appropriate elicitation techniques are critical in obtaining data for meaningful analysis. In the case of innovative designs, these problems are compounded by the fact that the experts know very little about the system that they are going to build.

The cost, C, of implementing a particular architecture is dependent on the set of architectural decisions $\{C(A_i)\}$. A project manager is expected to be able to estimate the cost of implementing various designs. By using traditional estimation techniques, an experienced project manager should not have difficulty coming up with a reasonably accurate estimate, but it may be a time-consuming and costly task—especially when several alternatives are being considered. To aid in this estimation exercise, rough implementation cost-characteristic curves that show how costs will behave with respect to each architectural decision may be of considerable help. For example, we may[2] observe that the cost of the system behaves like a step function every time a new processor is added within the architecture design. On the other hand, costs due to functional or implementation complexity may be smoother.

Determining the Stimulus/Response Characteristics from Architectural Decisions

One topic of keen interest among many in the software engineering community is the stimulus/response characteristics of architectures. the attribute-based architectural styles (ABAS) framework [Klein 99] aids the software architect in reasoning about the characteristics of a system that are quality-attribute specific. Using ABASs, the architect

[1] Here we consider the entire lifetime that the product is sold/maintained or supported.
[2] Whether the increment in cost is like a step function or just a minor increase depends on the total cost of the system and will have to be treated accordingly.

can map the relationship between A_i, the architectural decisions, and Q_i, the resulting quality attributes of the system.

Making Value Judgments Based on Stimulus/Response Characteristics

As mentioned earlier, value is derived from functionality *and* quality. Using Q_i, the stimulus/response characteristics of the system, the marketing managers and the various stakeholders can generate value judgments to obtain V_a. (A statement such as "We will have no users if serving a request takes more than 2 seconds" tells us that the value V_a of the system is approximately zero if the system's latency is greater than 2 seconds.) In this manner, for a particular system, the characteristic value curves can be elicited[1]. Some typical examples are shown in Figures 2a and 2b. These figures give the architect an idea of the sensitivity and criticality of the system's quality attributes.

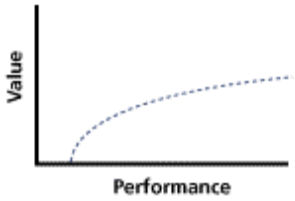

Figure 2a: Value vs. Performance

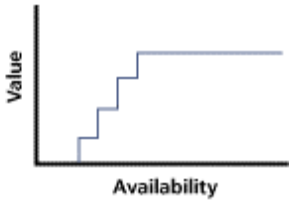

Figure 2b: Value vs. Availability

Importance of Choosing an Appropriate Time Scale

If we do not choose an appropriate time scale, the economics of a project cannot be properly assessed. The architects who design the architectures, the marketers who make the value judgments of functionality and stimulus/response, and the project managers who make the cost estimates for these architectures must have a common time basis for decision making. The system will end up badly designed if the time scales used by stakeholders are not the same[2]. Similarly, to assess its economic viability, an architecture that will form the basis for a product line will require a different time scale than would an architecture for a single product. Frequently the system's key stakeholders do not

[1] These curves will be approximate and elicited from experience. They will be used to give the architect a rough idea about the sensitivity of the market to certain quality attributes.

[2] The architecture for a product that is going to be used only for a 3-month period might be designed in an unmodifiable, non-scalable way, as opposed to the more modifiable approach that the architects would take if the system were to be used for 15 years.

properly understand or agree on what is expected of the product in terms of its lifetime and so cannot make these (cost-critical) decisions appropriately.

Developing Tools for Analysis

From this exercise we see that we will need analytic tools to do some meaningful analysis. First, we need a reasonable expert-elicitation technique through which we can obtain the time/cost/value structure characteristics of the system in which we are interested. The elicitation technique could be adapted from Morgan to suit cost estimation for software systems [Morgan 90]. This technique will be used in two places: for determining the value judgments of stimulus/response characteristics from stakeholders/marketers, and for determining the probable future costs of the system contingent on the architectural decisions.

Another important tool will be an appropriate method for discounting future costs. In traditional finance literature [Brearley 81], one uses the risk-free rate of return-on-investment to calculate the discount rate[1]. However, is this an appropriate instrument when analyzing software engineering projects? Considering the high-risk nature of most software projects, should the rate of return be higher than that used in financial markets?

Directions for Future Work

The development of a structure that incorporates the value judgment, cost-structure characteristics, and probable future costs with a consistent time scale is the essence of this work. Our areas of focus are in developing a good expert-elicitation technique as well as developing an appropriate method of discounting future costs.

Presently this work is being pursued as an extension to the ATAM. Cost will be considered as another tradeoff attribute. Additional stakeholders will be the marketing and sales managers who could give value judgments about the product and the expected lifetime of the product. We hope to validate this approach in the near future using case studies. An organization that is trying to reason about the various candidate software architectures for its software product and attempting cost-modeling at the design stage would be an ideal testbed for our approach. The benefit of our approach will be observed if it gives the software architect a business reason to make certain architectural decisions for the product—that is, the software architect could state in dollar terms[2] the reason for adopting a particular architectural choice.

---

[1] If "r" is the rate of return, then the discount rate $D=(1+r)^{-t}$ or e^{-rt}

[2] Even though this dollar value may be an expected or mean value.

References

[Brearley 81] R. A. Brearley, and S. C. Myers, *Principles of Corporate Finance*, McGraw Hill, 1981.

[Jones 98] T. Capers Jones, *Estimating Software Costs*, McGraw-Hill, 1999

[Kazman 99] R. Kazman, M. Barbacci, M. Klein, S. J. Carriere, and S. G. Woods, "Experience with Performing Architecture Tradeoff Analysis," Proceedings of ICSE 21 (Los Angeles, CA), May 1999, 54-63.

[Klein 99] M. Klein and R. Kazman, *Attribute-Based Architectural Styles*, CMU/SEI-99-TR-22, Software Engineering Institute, Carnegie Mellon University, 1999.

[Londeix 87] B. Londeix, *Cost Estimation for Software Development*, Addison-Wesley, 1987.

[Morgan 90] G. M. Morgan and M. Henrion, *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis*, Cambridge University Press, New York, 1990.

[Shaw 96] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

About the Authors

Jai Asundi is a doctoral candidate at the Department of Engineering and Public Policy at Carnegie Mellon University. His research interests lie in issues of cost modeling for software systems and issues of globalization of software development. He has a B. Tech. degree in chemical engineering from the Indian Institute of Technology, Bombay, and has worked for a software company prior to commencing his graduate studies.

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.

Mark Klein is a senior member of the technical staff at the SEI where he is a technical lead in the Architecture Tradeoff Analysis Initiative. He has more than 20 years of

experience in research and technology transition on various facets of software engineering and real-time systems. Klein's most recent work focuses on the analysis of software architectures. Klein's work in real-time systems involved the development of rate monotonic analysis (RMA), the extension of the theoretical basis for RMA, and its application to realistic systems. He has co-authored many papers and is a co-author of the RMA Handbook, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*.

COTS and Risk: Some Thoughts on How They Connect

David Carney

The focus of the past few columns has been on the various ways in which commercial off-the-shelf (COTS) products affect the way we develop systems: how we define requirements, evaluate products, and relate the activities of product evaluation and system design. This time I'd like to take a little step back, and focus on a different kind of issue that arises when an organization that is building a new system decides to "go COTS." I'd like to discuss a more programmatic topic, one that we at the SEI have had some interesting experiences with, and also describe some of our work in the area.

Risk: The Word of the Hour

The topic at hand is risk: its identification, evaluation, mitigation—all of those related things that we lump under the heading of *risk management*. The thrust of this column will be first to consider how, for a given project, the familiar notions of risk and risk management are affected by the presence of commercial software. Second, I will describe an approach that we have been developing that specifically addresses the question of risk evaluation for COTS-based programs.

Now, the importance of managing risk needs no advocacy here. Within the software community, it has been high on everyone's list for many years. In Barry Boehm's seminal work on "spiral" development, for instance, he presents a model that is inherently risk-driven. The SEI's Software Risk Evaluation (SRE) has been a useful risk management mechanism for dozens of projects. And mandates to do program management through risk management have now made their way into such high-level policy directives as the DoD 5000 series, Clinger-Cohen, Raines' Rules, and other such binding mandates.

And, as the saying goes, we are not alone. The predilection for seeing the world through risk-colored glasses is not unique to the software community; it is equally ubiquitous from a wider perspective as well. A recent national advertisement for General Motors automobiles stresses how "you will be risk-free" if you buy a GM car. Educational building designers focus on creating "risk-free environments" where students can learn. For many years now, every citizen of this land has become aware of the financial and credit agencies that increasingly invade our privacy to determine “how good a risk" each of us might be. And throughout our society as a whole, there is a growing perception that we are trying to nail down all-encompassing safety standards, tamper-proof bottles, child-proof everything, and so forth, presumably to reduce the daily risk of living our lives. (Other writers may wish to comment on whether such a goal is achievable, much less desirable. I am simply noting the societal trend.) With all of this interest in risk, I'm a

little surprised that the manufacturers of the old board game "Risk" have not seen fit to embark on a new advertising campaign. Given the current interest in the word, they could make quite a bundle.

But all of this furor notwithstanding, it is no less true that the central idea—that early in any enterprise we should identify those things that threaten our success, and either avoid them or somehow mitigate them—makes perfect sense in most things, and especially good sense in software development, a very risky enterprise indeed.

COTS: An Up-to-the-Minute Idea

It should be abundantly clear to readers of this column that the widespread use of commercial products in complex software systems poses many novel challenges to both developer and manager. We have, I hope, gained widespread agreement that "doing COTS" means doing some very different things. And there is ample evidence for the following assertion: the simplistic notion that one can easily "add COTS" to a traditional notion of system construction has brought grief to many programs. So as a community, we are now beginning to gain some clarity about the changes—some subtle, some drastic—that we must make when building COTS-based systems. But compared with the "build-from-scratch" paradigm, we are still in a fairly early stage of maturity, and most of us need a good deal more experience in many areas before we will have a firm understanding and broad expertise about "the COTS-based development paradigm" and what it means for system development.

These cautions are no less true when one deals with risk management. If one concurs that the impact of COTS is pervasive across the life cycle (and I hope that you do), and that good risk management entails analyzing and mitigating risks wherever they occur, then it is logical that risk management in COTS-based programs must accommodate the marketplace, whether subtly or drastically, just as everything else must. The changes may be various, though most often will involve COTS either bringing new sources of risk to a program, or exacerbating old forms of risk.

It is not difficult to conjecture some of the specifics. New sources of risk lurk wherever new entities appear. If we now bring many COTS vendors into the equation, then potential risks include the financial health [or “viability”] of those vendors, both now and at some point in the future. If dealing with vendors requires contractual stipulations, then novel risks include contract stipulations and guarantees about product commercialization, cooperation among vendors, licenses, ownership of throw-away prototypes, and many such unfamiliar issues.

Old and familiar risks tend to mutate, just as old and familiar practices do. Thus, we all think we know something about system design. But from a risk perspective, consider how

we now must deal with COTS-based design risks: must we really throw the design out the window if product *X* doesn't include PKI in its next release? What is the fallback? Do we go back to the old design we rejected last month? How do we plan to mitigate such a risk? And so forth, down the long litany of the way things are now, and the way things should be done differently with COTS.

CURE: A Solution for the Moment

The Software Engineering Institute has, over the past few years, been called in to examine a large number of DoD projects that were in varying stages of trouble. Some were in severe "crash-and-burn" mode, others were just beginning to nose toward the ground. But there were two common denominators in most of these troubled programs. One was the strong presence of COTS as a major factor in the systems being built. A second common factor in these programs was that all parties—on both the government and the contractor sides—were generally ignoring the particular risks that stemmed from COTS. Thus, after a number of these "red team" experiences, it became fairly clear that there was a pattern to the questions we were asking, and some common patterns to the behaviors we were witnessing.

We found ourselves repeating the same list of "didn't anyone think of *X*" questions, where *X* would be something like "asking the vendor about his market share," or "asking why the product needed to be modified." We found some common areas where significant danger was waiting for programs—product modification being a major one—and also found that many program managers were simply unfamiliar with these dangers, and thus simply hadn't thought to ask some of these questions.

The result is that we developed a mechanism that would fill this specific need, at least during the present period when there is so little shared experience with using COTS. We distilled the aggregate experience from those "red teams" into a form of COTS-centric risk evaluation, to be performed early in the acquisition process. Some persons have called it a front-end "red team," which is a convenient, though perhaps simplistic, way to think about it.

This mechanism is called a "COTS Usage Risk Evaluation," with the imposing acronym of "CURE[SM]." (More later on the implications of that acronym.) CURE is quite different from the "risk evaluation" as found in a traditional SRE. For one thing, an SRE has the philosophical goal of motivating the personnel of a program to understand the general nature of risk, and to change their behavior, learning to manage it on an ongoing basis. CURE has a different philosophical goal. It is a diagnostic tool aimed at a particular and bounded target, namely the presence of COTS products and the risks that accompany their use.

CURE is intended to be used on any of the participants in a program, whether on the acquiring side or the contracting side. This is a direct result of our "red team" experiences, where both government and contractor exhibited the behaviors that led to the problems. CURE is also designed to be aimed at key personnel: the actual program manager, the contractor's lead architect (or chief engineer, or whatever the job title actually is), and the contractor's project manager. In brief, CURE seeks to find the key decision makers of a project, and to understand their individual awareness of COTS-related issues.

CURE consists of an initial data-gathering phase, an onsite visit, and preparation of a detailed evaluation report. The central instrument for CURE is an extensive questionnaire, used in both the initial data gathering and the onsite visit. The questionnaire covers the following topic areas:

General Topics:

- system description
- management readiness
- technical readiness

Business Topics

- contractual issues
- vendor and supplier relationships

Infrastructure Topics:

- standards
- process
- environments

Lifecycle Topics:

- COTS product evaluation
- system design
- integration
- testing
- maintenance

First, the questionnaire is sent to the site to be filled out and returned to the evaluation team. This provides the evaluation team with an initial view of the program, and also indicates to the program manager, et al., what kind of data the evaluation team is searching for. During the onsite visit, the topics in the questionnaire are revisited in

greater detail. In addition to the raw data that the questionnaire seeks, each question also has a number of optional discussion topics, which are the basis for longer discussions during the onsite visit.

The following excerpt shows the format and content of one of the CURE questions:

11.3 COTS modification

Describe the degree to which the system will require program-specific modification, tailoring, extensions, or enhancements to COTS products.

Identify aspects of the system design that are (or are expected to be) dependent on modifications to specific COTS products.

Potential discussion topics for onsite interview:

Strategy for product replacement if necessary

Decision factors that indicate modification is necessary

How the level of necessary modification was determined

Estimates for the cost and schedule of making theses modifications

Plans for sustainment of the modifications

After the onsite interview, the evaluation team prepares a report detailing the major COTS-related risks to the program. Each statement of risk includes the dangerous condition that exists (or will exist), the factors that led the team to this analysis, the resulting consequence, and the overall criticality of that consequence to the program. Perhaps the most important part of the report is that, for each risk, we also suggest one or more mitigations. This report is sent to the evaluation site within 10 working days after the onsite interview.

Results of CURE

As noted above, the acronym is somewhat misleading because this mechanism is not really a "cure" so much as it is a diagnosis. But that slight cheating aside, we have found CURE to be a valuable and powerful tool. In the programs that have had CUREs so far, we have generally found roughly a dozen critical risks, some of which were surprising to the personnel involved, others of which were known but underestimated. Indeed, one very useful result of CURE, when done on both the government and contractor side, is to

unearth misunderstandings about the very same issues from the two sides, a common problem found when rights to products are at stake. ("But I thought *we* had bought all rights to the source code! Didn't the contract define that?")

At the moment, we are revising the questionnaire based on the experiences gained during early trials of the mechanism. We are also in discussion with one large DoD agency concerning transition of CURE for their use. As we perform more evaluations, we hope to gather some firmer data about its efficacy, both in predicting risks to programs as well as in giving them some management assistance in mitigation of those risks. When that data is available, we will report it, either in this space or in an SEI technical report. 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.

Survivability Blends Computer Security With Business Risk Management

Howard F. Lipson, David A. Fisher

In recent years, there have been dramatic changes in the character of security problems, their technical and business contexts, and the goals of system stakeholders. As a consequence, many of the assumptions underlying traditional security technologies are no longer valid. In particular, a new fundamental assumption is that any individual component of a system can be compromised by attacks, accidents, or failures. To ensure that mission-critical functions are sustained and essential services are delivered despite the presence of attacks, accidents, or failures, a *survivability* perspective on security practices is needed.

Survivability from a Business Perspective

Many businesses have contingency plans for how to handle business interruptions caused by natural disasters or accidents. Although the majority of cyber-attacks are relatively minor in nature, a cyber-attack on an organization's critical networked information systems has the potential to cause severe and prolonged business disruption, whether the business has been targeted specifically or is a random victim of a broad-based attack. If a cyber-attack disrupts critical business functions and interrupts the essential services that customers depend on, the survival of the business itself is at risk. Moreover, a business disruption caused by a cyber-attack will likely be seen by customers as a sign of incompetence. Unless the cyber-attack is widespread and well publicized, no customer sympathy will be forthcoming.

Survivability is an emerging discipline [ISW 97, ISW 98] that blends computer security with business risk management for the purpose of protecting highly distributed information services and assets. A fundamental assumption is that no system is totally immune to attacks, accidents, or failures. Therefore, the focus of this new discipline is not only to thwart computer intruders, but also to ensure that mission-critical functions are sustained and essential situation-dependent services are delivered, despite the presence of cyber-attacks. Improving survivability in the presence of cyber-attacks also improves the organization's capacity to survive accidents and system failures that are not malicious in nature.

Traditional computer security is a highly specialized discipline that seeks to thwart intruders through technical means that are largely independent of the domain of the application or system being protected. Firewalls, cryptography, access control, authentication, and other mechanisms used in computer security are meant to protect an underlying application in much the same way regardless of the specific application being

protected. In contrast, survivability has a very sharp mission focus. Ultimately it is the mission that must survive, not any particular component of the system or even the system itself. (The mission must go on even if an attack causes significant damage to—or even destroys—the system that supports the mission.) This focus on mission survivability rather than on system protection is the most radical paradigm shift that is occurring as the new discipline of information survivability emerges.

Survivability solutions are best understood as risk-management strategies that first depend on an intimate knowledge of the mission being protected. Risk-mitigation strategies first and foremost must be created in the context of a mission's requirements (prioritized sets of normal and stress requirements), and they must be based on "what-if" analyses of survival scenarios. Only then can we look toward generic software engineering solutions based on computer security, other software quality-attribute analyses, or other strictly technical approaches to support the risk-mitigation strategies.

Consider the analogy of a village farmer with the mission of supplying food to a village. The farmer may have a fence around the crops to keep out deer, rabbits, and other intruders (traditional security). The farmer may have an irrigation system to be used in the event of insufficient rainfall (redundancy). He or she may plant a variety of crops so that even if environmental conditions such as pests adversely affect one crop, others will survive (diversity). All of this is well and good. But even if all the crops fail and no food is grown, the mission can still succeed if the farmer has an alternate strategy based on the mission of providing food—*not* necessarily growing food using the local ecosystem. If the crops fail, the farmer may turn to hunting or fishing to provide the life-sustaining mission fulfillment that fellow villagers depend on. Is hunting a security, reliability, or fault-tolerance strategy? No, it is outside the system for growing food. This is a risk-management strategy that can only be formulated with an intimate understanding of the mission that must survive. Detailed technical expertise on fence-building, or even agriculture, is helpful but inadequate compared to strategies based on an intimate knowledge of the mission requirements.

Survivability depends not only on the selective use of traditional computer-security solutions, but also on the development of effective risk-mitigation strategies based on scenario-driven "what-if" analyses and contingency planning. "Survival scenarios" positing a wide range of cyber-attacks, accidents, and failures aid in the analyses and contingency planning. However, to reduce the combinatorics inherent in creating representative sets of survival scenarios, these scenarios focus on adverse effects rather than causes. Effects are also of more immediate situational importance than causes because you will likely have to deal with (and survive!) an adverse effect long before determining whether the cause was an attack, an accident, or a failure.

Contingency (including disaster) planning requires risk-management decisions and economic tradeoffs that only executive management can make (preferably with guidance

from technical experts). Survivability depends at least as much on the risk-management skills of an organization's management as it does on the technical expertise of a cadre of computer security experts. (Here we are not referring to an abstract technical skill in the science of risk management, but rather to the ability to manage risk in the context of the specific business mission and goals.) Business risk management is arguably the primary function of executive management. The role of the experts in security, the application domain, and other technically relevant areas is to provide executive management with the information necessary to make informed risk-management decisions. Thus, the preparatory steps necessary for survivability must be taken by an organization as a whole, rather than by security experts alone.

Let's consider the Galaxy-4 satellite that spun out of control on May 19, 1998, interrupting up to 90% of the pager service in the United States, along with some television network feeds and some credit card verification services. Even though a cyber-attack was not to blame (though "an international hacker attack" was on an early list of speculative causes), the example is quite illustrative. In fact, the cause (or at least a partial cause—crystals forming on tin-plated relay contacts and an unexplained failure of a backup system) was not determined until long after service was restored [Reuters 98].

Dealing with adverse events such as this one, without waiting for a definitive determination of the cause, is central to the survivability paradigm. Successful handling of such events depends far more on prudent risk management and contingency planning by executive management than on any specific technical approach by security or other experts. For instance, a "perfect" technical solution (i.e., having a diversely redundant, immediately available backup satellite) would be economically unfeasible. The practicality of many technical solutions can only be evaluated in the full business context. Executive management, through its contingency planning, would consider business solutions that might transcend purely technical solutions. One approach would be to have an agreement in place with another communications company to provide the needed capacity on, say, six hours' notice (with the backup company dumping its own lower-priority customers) in exchange for an annual fee.

An alternate approach—using lawyers rather than technologists—would be to have a disclaimer in the contract agreement with customers telling them that the customer must bear the risk of service outages. This would put the customers on notice that they need to prepare to provide their own redundancy, whereas in the previous approach the service provider took care of redundancy through an agreement with an alternate provider. Because it raises customer awareness of some of the risks inherent in the delivery of service and possibly increases the perceived value of uninterrupted service, the "legal disclaimer" approach might even generate some customer interest in asking the original service provider to provide redundancy for an additional fee. The "legal disclaimer" approach is not one that technical experts would likely come up with, but it is quite effective in assuring the survivability of the business mission and goals. As this example

illustrates, the risk-management viewpoint supports an "economics of survivability" that allows businesses to successfully prepare for and overcome the adverse effects of cyber-attacks, accidents, and failures with approaches that can transcend those offered by technical experts alone.

Contrast this new perspective with current practice. Upper management's primary decision-making role, from a traditional security viewpoint, is predominantly to determine how much direct funding and other resources to grant to the organization's security experts for the rather loosely defined purpose of "beefing up security" to some vaguely articulated industry-standard level of practice. In the minds of management, the perceived link between security funding and the business mission (and the business bottom line) is tenuous at best. "If I spend more money on computer security, my risk of intrusion will likely go down. But will this reduce any significant risks to my business mission? What risks will be reduced, and by how much?" With no clear benefit visible to management, the resulting security funding is typically inadequate to meet even the limited technical goals of the security experts. For the most part, what is sorely missing is an in-depth analysis of threats to the organization's mission and a corresponding cost–benefit analysis for risk-mitigation strategies and contingency planning. The computer-security experts, isolated from management's intimate understanding of the business mission, are in no position to perform the necessary threat analyses, except from the narrow perspective of their technical specialties.

As an example, consider the new government programs that are meant to assure that our nation's critical infrastructures will continue to operate despite cyber-attacks, accidents, or failures. Government concern for critical infrastructure assurance [PCCIP 97] is helping to fuel the current interest in survivability, but this interest is not being driven by the businesses (such as those in energy, transportation, banking, and telecommunications) that would benefit from such protection. The government is asking industry to participate in critical-infrastructure assurance programs, with the motivation that these programs are in the best interests of the nation, the industries, and their customers. But none of these communities are willing to pay for the increased costs. Real investment in critical-infrastructure protection will occur only when executives understand that these changes are essential to their competitiveness and profitability. Unfortunately many of the businesses involved see these programs as mandating technical solutions that would be at odds with their customers' needs and their own profitability. Greater awareness is needed of the business risk-management aspects of survivability, so that the organizations that operate our nation's critical infrastructures would be motivated by self-interest to assure their own survivability. Critical-infrastructure assurance can then be based on risk-management tradeoffs that depend on overall business missions and goals, not solely on technical fixes that are independent of those goals.

Summary

There has been a revolutionary technical shift in business applications from stand-alone, closed systems over which organizations exercised complete control, to highly distributed, open, COTS[1]-based systems over which only very limited control and limited insight are possible. Not only are most Internet services outside of the control of the businesses that use them, but so is the functionality and software quality attributes of the COTS-based software used to build business applications. This technical shift has taken us so far that we can no longer solve security problems entirely in the technical domain.

From the traditional computer-security perspective, executive management has never been sufficiently engaged. The security experts simply present a bill or funding request to management for generic technical solutions, independent of threat analyses that are specific to the mission being protected.

Upper management must be concerned with threats to the business mission and must be intimately involved in formulating mission-specific, risk-mitigation strategies. Moreover, technical experts need to be aware of the business issues that lead to the technical issues that they face. Only then can they contribute effectively to the risk-management approaches that are needed to assure survivability of highly distributed mission-critical applications, operating in unbounded domains, in the face of cyber-attacks, accidents, and system failures.

For Additional Information

This column is based on the following paper, which contains additional information about this topic.

H. F. Lipson and D. A. Fisher, "Survivability—A New Technical and Business Perspective on Security," *Proceedings of the 1999 New Security Paradigms Workshop*, September 21-24, 1999, Association for Computing Machinery, 1999.

References

[ISW 97] *Proceedings of the 1997 Information Survivability Workshop*, San Diego, California, February 12-13, 1997, Software Engineering Institute and IEEE Computer Society, April 1997. Available at: http://www.cert.org/research/

[1] Commercial off-the shelf

[ISW 98] *Proceedings of the 1998 Information Survivability Workshop*, Orlando, Florida, October 28-30, 1998, Software Engineering Institute and IEEE Computer Society, 1998. Available at: http://www.cert.org/research/

[PCCIP 97] Presidential Commission on Critical Infrastructure Protection, *Critical Foundations — Protecting America's Infrastructures*, The Report of the Presidential Commission on Critical Infrastructure Protection, October 1997.

[Reuters 98] Reuters, "Pager Glitch Cause Lost in Space," *Wired News*, Wired Digital Inc., August 11, 1998. Available at: http://www.wired.com/news/news/technology/story/14355.html

About the Authors

Howard F. Lipson has been a computer security researcher at the SEI's CERT Coordination Center for nearly eight years. He has played a major role in extending security research at the SEI into the new realm of survivability. His research interests include the design and analysis of survivable systems, survivable systems simulation, and critical infrastructure protection. Lipson has been a chair of two IEEE-sponsored workshops on survivability. Earlier, he was a computer scientist at AT&T Bell Labs, where he did exploratory development work on programming environments, executive information systems, and integrated network management tools. He holds a PhD in computer science from Columbia University.

David A. Fisher is currently leading a research effort in new approaches for survivability and simulation in information-based infrastructures at the SEI's CERT® Coordination Center. From 1973-75, Fisher served as program manager in the Advanced Technology Program (ATP) at the National Institute of Science and Technology (NIST), where he developed and managed a major initiative in component-based software and began an initiative in learning technology. Fisher has more than 60 publications in the areas of information survivability, algorithms, component-based software, programming languages, compiler construction, and entrepreneurial development in the software industry. He earned a PhD in computer science at Carnegie Mellon University, an MSE from Moore School of Electrical Engineering at the University of Pennsylvania, and a BS in mathematics from Carnegie Institute of Technology.

Justifying a Process Improvement Proposal

Watts S. Humphrey

My December 1999 column described how to make the strategic case for process improvement. In this column I provide an example of how to do this. This column thus assumes that you have the ear of a manager or executive who thinks strategically and will consider investments that will likely take a few years to pay off. In the next column, I will talk about dealing with tactically focused managers.

The Financial Justification Process

The financial justification process has five phases:

Phase 1 Decide what to do.

Phase 2 Estimate the likely costs.

Phase 3 Estimate the likely improvement benefits.

Phase 4 Produce the improvement proposal.

Phase 5 Close the deal.

The December 1999 column generally discussed these steps. This column walks you through a hypothetical case study in which Tom Jones develops a proposal to introduce the Team Software Process (TSP)[SM].

Phase 1: Decide What to Do

Tom reviewed the situation in his organization and found that management's top priority was to reduce development cycle time. He decided to do this by introducing the TSP. He also talked to experts about the TSP introduction strategy and found that it had the following seven steps:

Step 1 Hold an executive seminar for selected top managers and executives from the division or laboratory. Tom estimated that there would be 20 attendees.

Step 2 Give a half-day planning session to determine the improvement plan. Tom assumed that 10 of the first-day attendees would participate.

Decision 1 Tom assumed that these first two phases would be successful but decided to include a decision step to reduce the required initial commitment.

Step 3 Train the involved managers in the Personal Software Process (PSP[SM]) and TSP management methods. Tom planned to include the team leaders, the managers of the team leaders, and several other managers below the executive level. He assumed there would be 10 managers in this course.

Step 4 Provide PSP training to all the engineers who will be on the teams. Tom assumed there would be 2 teams with 8 engineers per team, or 16 engineers.

Step 5 Provide general PSP training to any team members who are not programmers. This would be systems, hardware, or test personnel, for example. He assumed that the first two TSP teams would be software only and that there would be four system-requirements team members in this category.

Step 6 Tom planned to train two engineers to be PSP instructors so they could support the two TSP teams, handle training for any team turnover replacements, and support further TSP introduction. These two instructor candidates would also attend PSP training in step 4, bringing that total up to 18.

Step 7 The final introduction step Tom planned was to launch the two TSP teams.

Decision 2 Assuming that these initial team launches were successful, Tom planned to ask management to proceed with broader TSP introduction.

Phase 2: Estimate the Likely Costs

After Tom defined the proposed introduction program, he estimated its costs in four parts:

1. Labor costs
2. Internal support costs
3. Consulting, training, and external support costs

4. Lost opportunity costs

Estimating the Labor Costs

For the labor costs, Tom estimated the number of people to be involved in each of the introduction steps as shown in Tables 1 and 2. Because the TSP launches in step 7 would be part of the project, however, he did not count them as training time.

Table 1. PSP and TSP Introduction Program Training

Step	Item	People	Prep. Time	Class Days	Engineer Days	Manager Days
1	*Executive Seminar*	20		1.0		20
2	*Planning Session*	10		0.5		5
3	*Manager PSP Training*	10	0.3	4.0		43
4	*PSP Course I & II*	18	2.5	7.0+7.0	297	
5	*General PSP Course*	4	0.5	3.0	14	
6	*Instructor/Coach Courses*	2		10.0	20	
	Totals				331	68

Next, Tom checked with the financial people and found that the cost for a day of engineering time was about $1,000 and that a manager or executive day cost about $2,000. While these rates seemed high to him, finance explained that they included all the costs for overhead, support, vacation, medical benefits, sick time, workers' compensation, insurance, retirement, Social Security, and taxes. Using these numbers, Tom calculated that the labor costs to train the two TSP teams and their managers would be $331,000 + $136,000 = $467,000.

Table 2. Total TSP Five-Year Costs

Cost Item	Cost calculation	Total cost
Internal introduction costs	$1,000*331 engineering days $2,000*68 manager days	$467,000
TSP coaching costs	2 engineers*40 days*1 year	$80,000
External introduction costs	25% to 75% of internal costs	$136,750 to $410,250
Total one-year costs		$683,750 to $957,250
TSP coaching costs	2 engineers*40 days*4 years	$320,000
Turnover training	3 engineers*14 days*4 years	$168,000
Total five-year costs		$1,171,750 to $1,445,250

Support Costs

Tom found that each TSP team would need coaching support of about 20% of the time of a TSP instructor/coach. These costs would add about 80 engineer days per year or $80,000 a year.

Finally, any new or replacement engineers on the teams would have to be trained. Assuming an annual turnover of 20%, that would be about three engineers to train per year at 14 days of training each, or another 42 days of training and $42,000 per year.

External Costs

The costs for the external instructors and consultants for any improvement program are typically fairly large, but they are generally much smaller than the labor costs. These external costs would include delivering the courses listed in Table 1, launching and supporting the initial TSP teams, several team relaunches, and occasional consultation and assistance.

Without going out for external quotes, Tom assumed that the external support costs for this initial effort would add between 25% and 75% to the first-year labor costs, with the level of introduction cost determined by how rapidly management wanted to introduce the TSP.

Lost Opportunity Costs

Tom realized that while the engineers and managers were being trained they would not be developing or supporting products. To account for these costs, he would reduce the anticipated first-year cycle-time improvement by the three-week engineer training time.

Phase 3. Estimate the Likely Improvement Benefits

In estimating the improvement benefits, Tom found data on the benefits that other organizations had enjoyed with the TSP, and he also obtained data on the current performance of his organization. Finally, he used these data to estimate the likely improvement benefits for his organization.

Identify Available Improvement Data

Tom learned that Hill Air Force Base, on its first TSP project, increased productivity by 123% over the same team's prior project [Webb]. Hill AFB also cut test time from 22% to 2.7% of development time, or a reduction of 88%.

He also found a presentation by Teradyne on its TSP results [Musson]. Through the use of the TSP, Teradyne cut final test and field defects from a rate of 20 defects per thousand lines of code (KLOC) to 1 defect/KLOC. Historically, final test and field defects had cost Teradyne an average of 12 engineering hours per defect to find and fix.

By using the TSP, Tom also learned that Teradyne had reduced engineering and customer-acceptance test time from nine months for an earlier project to five weeks for the TSP. Engineering and acceptance test defects were cut from 5 defects/KLOC to 0.02 defects/KLOC, with no further defects found by the customers.

In addition, he found that a Boeing TSP project had cut test defects by 75% from the prior project average and reduced system test time by 96% [Vu].

Organizational Performance Data

To calculate the TSP benefits for his organization, Tom next needed data on how much time these development groups spent in test, the level of defects in the various test phases, and the cost of diagnosing and fixing each defect. With these data, he could then estimate the likely savings from introducing the TSP.

Tom assumed that the 16 engineers on the 2 trial projects would develop a total of about 80,000 lines of code in 12 months. He also found that the time typically spent in integration and system test was currently about 40% of the development schedule and

that the defect levels in test and field use were much like those at Teradyne, or 20 defects/KLOC. He assumed that these 20 defects were split with 10 in integration test, 5 in system test, and 5 after product shipment. He also assumed that the engineering cost to diagnose and fix these defects was about 1.5 days per defect.

Estimate the Likely Cost Savings

As shown in Table 3, Tom now estimated the likely savings. First, for the reduction in test defects, he projected that the integration and system test defect/KLOC would be reduced from 15 to 1, for a savings of 14 defects/KLOC. For the total 80,000 lines of code planned for these two projects, he calculated that this would save 14*80 = 1,120 test defects. At a cost of 1.5 engineering days for each defect, this would be a reduction of 1,680 engineering days or $1,680,000.

Table 3. Estimated Savings

#	Item	Change	Days	Savings
#1	Integration/system test defects	- 1,120 defects	1,120*1.5=1,680	$1,680,000
#2	Test time	- 4.1 months	4.1*21*16=1,378	$1,378,000
#3	Maintenance costs	-398 defects	597	$597,000
#4	First-year savings (#2+#3)			$1,975,000
#5	First-year costs (Table 2)			$957,250
#6	First-year ROI (100*#4/#5)			206%
#7	Five-year savings (#4*5 years)			$9,875,000
#8	Five-year costs (Table 2)			$1,445,250
#9	Five-year ROI (100*#7/#8)			683%

To check these savings, Tom also looked at test-time reductions. The Boeing results showed a test-time reduction of 96% and Hill AFB reported an 88% reduction. Tom assumed that the TSP would reduce his organization's integration and system test time by

85%, or from 4.8 months to 0.7 months for a 12-month project. This 4.1-month savings, at 21 working days per month and 16 engineers, came to a savings of 1,378 engineering days. At the typical engineering day rate of $1,000, the test time savings were then $1,378,000. Since this estimate was a little lower than the $1,681,000 savings based on defect reduction, he decided to use the lower number in the justification calculations.

In addition, Tom also felt that there would be a maintenance cost reduction. For field and customer defects, the Teradyne data showed a reduction from 5 defects/KLOC to 0.02 defects/KLOC. For the 80,000 lines of code planned in his organization, he estimated a reduction of 398 customer-reported defects. At 1.5 engineering days each, this would be a maintenance-cost savings of $597,000, for a total savings of $1,975,000 in one year. He then compared this to the maximum one-time improvement cost of $957,250 to give a one-year return on investment of 206%.

Finally, Tom assumed that the initial two TSP teams would continue to use the TSP on subsequent projects. For the next four years, additional costs would be needed to cover training for engineering turnover ($168,000) and 20% of the time for the two PSP instructor/coaches ($320,000). The five-year savings would then be $9,876,000 and the five-year improvement cost would be $1,445,250, for a return on investment of 683%.

Cycle-Time Benefits

Next, Tom used the reduction in test time to estimate cycle-time improvement. With the assumed 12-month project schedule and 40% of the schedule spent in test, normal testing time would be 4.8 months. By assuming an 85% reduction in test time, he estimated that the 4.8 months would be reduced to 0.7 months. Thus, the typical 12-month schedule would be cut to eight months, or about a 32% cycle-time reduction. For the first year, he also reduced this 4.1-month cycle time improvement by the three-week initial team training time, for a net of 3.35 months schedule savings.

Phase 4. Produce the Improvement Proposal

At this point, Tom had completed the estimating work and needed to produce a brief management presentation. He decided on the following seven-part outline:

Presentation *Part 1*	Opening summary. On one chart, he briefly summarized the problem, how he proposed to address it, and the likely benefits.	
	The problem	To improve cycle time and not increase costs
	The solution	Introduce the Team Software Process (TSP)

	Likely benefits	A 32% reduction in cycle time
		A $2 million one-year savings, $10,000,000 in five years
		A $950,000 one-time introduction cost
		About $500,000 in subsequent four-year support costs
		A one-year return on investment (ROI) of 206%
		A five-year ROI of 683%

Presentation *Part 2*

The proposal. He briefly described the proposal.

He first asked for a minimum commitment to steps 1 and 2 of the introduction program.

Second, assuming that the first two steps were successful, he proposed to proceed with the two-team TSP pilot program.

Third, after the two teams were underway and the early experience was satisfactory, he planned to proceed with broader TSP introduction.

Presentation *Part 3*

Description of the TSP. As backup, he prepared a brief description of the TSP and its objectives.

Presentation *Part 4*

Summary of the TSP benefits obtained by other organizations. Also as backup, he prepared a summary of the available data.

Presentation *Part 5*

Summary of the introduction plan. As backup, he made a list of the principal phases and decision points in the introduction plan.

Presentation *Part 6*

Summary of the estimated cost savings. As backup, he made a summary of the cost savings calculations with the key assumptions and supporting data. He also reviewed these figures with finance before the presentation.

Presentation *Part 7*

Summary of the estimated cycle-time reduction. As backup, he included a summary of the cycle-time improvement calculations with the key assumptions and supporting data.

Phase 5. Close the Deal

Tom's final step was to make the presentation and get the order.

Closing Comments

While this example is for a specific process-improvement method, the principles are quite general. To relate this example to the discussion in the December column, the five phases in this example relate to the topics in the December column as follows:

Phase 1 Decide what to do:

- Clearly define what you propose.
- Understand today's business environment.
- Identify the executive's current hot buttons.
- Make an initial sanity check.

Phase 2 Estimate the likely costs:

- Start the plan with two or three prototypes.
- Estimate the one-time introduction costs.
- Determine the likely continuing costs.

Phase 3 Estimate the likely improvement benefits:

- Document the available experience data.
- Estimate the expected savings.
- Decide how to measure the actual benefits.
- Determine the improvement's likely impact on the executive's current key concerns.
- Identify any other ways that the proposed improvement could benefit the business.

Phase 4 Produce the improvement proposal:

- Produce a presentation to clearly and concisely give this story.

Phase 5 Close the deal.

I hope this example will help you make your own business case. If you have questions, I suggest you look at the December 1999 column, which contains a more generic discussion of these same topics.

Stay tuned in

In the next issue, we will discuss the issues of convincing tactically focused managers and executives to start a process-improvement program. Following that, a subsequent column will deal with how to move from a tactically based to a strategically based improvement program.

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful comments and suggestions of Sholom Cohen, Frank Gmeindl, Julia Mullaney, Jim Over, Mark Paulk, 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

[Musson] Robert Musson, presentation at the 1999 Software Engineering Institute Symposium, Pittsburgh, PA, August 30 to September 2, 1999.

[Vu] John Vu, presentation to the U.S. Department of Defense on the Boeing process-improvement program.

[Webb] Dave Webb and W. S. Humphrey, "Using the TSP on the TaskView Project," Crossstalk, vol. 12, no. 2, February, 1999, pp. 3 - 10.

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.