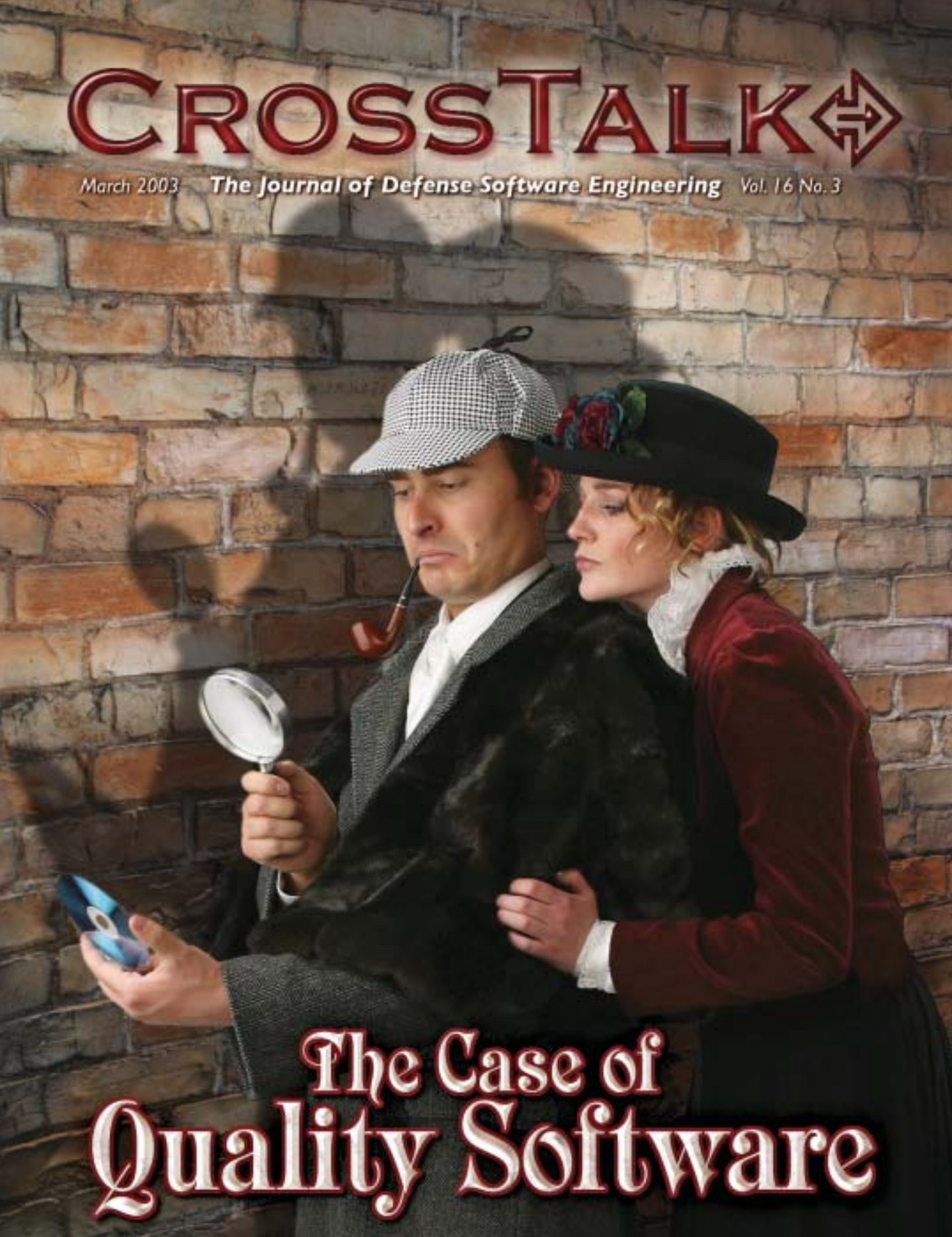
CrossTalk

March 2003 The Journal of Defense Software Engineering Vol. 16 No. 3



The Case of Quality Software

Quality Software

ON THE COVER
Cover Design by Kent Bingham. Models: Brice and Heidi Anderson. Photo by Kevin Dilley.

Best Practices

Open Forum

Online Article

Departments

CrossTalk

SPONSOR *Lt. Col. Glenn A. Palmer*
PUBLISHER *Tracy Stauder*
ASSOCIATE PUBLISHER *Elizabeth Starrett*
MANAGING EDITOR *Pamela Bowers*
ASSOCIATE EDITOR *Chelene Fortier*
ARTICLE COORDINATOR *Nicole Kentta*
CREATIVE SERVICES COORDINATOR *Janna Kay Jensen*

PHONE (801) 586-0095
FAX (801) 777-8069
E-MAIL crosstalk.staff@hill.af.mil
CROSSTALK ONLINE www.stsc.hill.af.mil/crosstalk
CRSIP ONLINE www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 21.

Ogden ALC/MASE
7278 Fourth St.
Hill AFB, UT 84056-5205

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CROSSTALK editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CROSSTALK.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

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

STSC Online Services: www.stsc.hill.af.mil
Call (801) 777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CROSSTALK available free of charge.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

We Need the Right Tools for Quality Software

Several years ago, I was supporting a software development effort at a commercial company. My boss decided to add to my responsibilities by having me develop part of the software using a database manager. I did not have any experience with database development, and my management didn't provide me with formal training. However, they did provide me with tutorial textbooks, time to learn, and a customer support contract with the software company so that someone could help me as I needed questions answered. The end result was my software delivered on time, on budget, and with no errors reported from the customer.

Not long after that, I was hired by another organization that wanted me to support their database development. However, they wanted me to use an application of the database manager that I had never used before. I told them that I had never used this application, but I knew about it and wanted to learn it, so they promised to give me that opportunity. I naively assumed that meant the same arrangement as my previous employer: books, time, and customer support. I was wrong. I was given the books and told that I was expected to learn the application on my own time (in addition to the eight hours of uncompensated overtime that they also told me I was expected to work each week). No one else in the organization was familiar with this application, so I also didn't have anyone to answer questions. The results weren't as successful as at my previous employer. I finally left the company while the project was behind schedule, and we didn't know when we would be finished; I don't know what the end result was.

The articles in this month's issue provide some good insights on things that can be done to help an organization develop quality software. I would like to add to these ideas by reminding managers to provide their people with the tools needed for the job. By tools, I'm not recommending more software packages; I'm recommending training, technology, and appropriate time.

Getting to this month's articles, David N. Card starts us out with *Managing Software Quality With Defects*. Card is a leader in the software community on measurement, and he shares his insights for quality measurements in this article. I believe in the concept that if you can't measure it, you can't manage it. If we want quality in our software, then we need to track and manage the quality while the software is being developed.

Next, Dr. Kenneth D. Shere gives an overview of Lean Six Sigma (LSS) in *Lean Six Sigma: How Does It Affect the Government?* He starts his article by discussing what LSS is, then goes on to discuss the benefits realized by several organizations using LSS, and how it can be used for acquisition.

Gary E. Mogyorodi used to work with Richard Bender, the developer of requirements-based testing. Mogyorodi shares the knowledge he gained from this association in *What Is Requirements-Based Testing?* If you would like the opportunity to get more information directly from Mogyorodi, he will be speaking at this year's Software Technology Conference, held April 28-May 1 in Salt Lake City. Don O'Neill also shares his insight in *Determining Return on Investment Using Software Inspections*. If you're not already using software inspections, you're missing the boat; I hope this article will motivate you to start.

The supporting articles in this issue should also provide some useful pointers. Dr. Randall W. Jensen shares his first encounter with pair programming in *A Pair Programming Experience*. Jensen discusses how this experiment was arranged, and how it turned out. We also share a discussion of software architectures in Ingmar Ögren's *Clarify the Mission: A Necessary Addition to the Joint Technical Architecture*. Lastly, Gary E. Mogyorodi presents 20 questions used to determine the maturity of an organization's quality assurance and testing environments in his online article *Let's Play 20 Questions: Tell Me About Your Organization's Quality Assurance and Testing*.

I hope the managers and acquirers who read CrossTalk are using the insights shared by software industry leaders to enable their people and contractors to develop the software they are capable of developing. This requires working with those developers to determine realistic delivery schedules, providing resources for necessary training, and providing other support needed to get the job done right the first time.

Elizabeth Starrett

Elizabeth Starrett
Associate Publisher

Managing Software Quality With Defects[1]

David N. Card
Software Productivity Consortium

This article describes two common approaches to measuring and modeling software quality throughout the software life cycle so that it can be made visible to management. Both approaches involve developing a life-cycle defect profile, which serves as a "quality budget." This article also provides actual examples using each approach.

Many factors contribute to an increased practical interest in managing software quality. This means treating software quality as a key dimension of project performance, equal to cost (effort) and schedule. Corporate initiatives based on the Capability Maturity Model® (CMM®) [1], CMM Integration℠ (CMMI℠) [2], and Six Sigma [3] provide some examples of forces promoting an interest in quality as a management concern.

General management activities include planning, monitoring, and directing. In order to manage quality, it must be planned; accomplishment of the plan must be tracked, and appropriate corrective action must be taken as necessary. Nearly all projects establish budgets for effort and/or cost so that these dimensions can be managed. These budgets are plans for the expenditures of labor and/or dollars during the life of the project. Budgets typically identify planned total expenditures as well as expenditures during specific intervals such as life-cycle phases or months. Managing quality also requires establishing a budget for quality.

This article presents a simple approach to measuring and modeling software quality across the project life cycle so that it can be made visible to management. Next in the article are examples of applying this measuring and modeling approach in real industry settings. Both of the examples presented come from CMM Level 4 organizations. Whether or not the CMM or CMMI explicitly requires this type of analysis is beyond the intended scope of this article. More importantly, the approach has been shown to be practical and useful to project managers.

Software Quality and the Defect Profile

There are many views of software quality. The ISO/IEC 9126 [4] defines six:

- Functionality.
- Efficiency.
- Reliability.
- Usability.
- Maintainability.
- Portability.

Some of these quality factors are difficult to measure directly. Intuitively, the occurrence of defects is negatively related to functionality and reliability. Defects also interfere, to some degree, with other dimensions of quality. Both of the approaches discussed here involve developing a life-cycle defect profile. This defect profile serves as a *quality budget*. It describes planned quality levels at each phase of development just as a budget shows planned effort (or cost) levels. Actual defect levels can be measured and compared to the plan, just as actual effort (or cost) is compared to planned effort (or cost). Investigating departures from the plan leads to corrective actions that optimize project performance.

Software development consists of a series of processes, each of which has some ability to insert and detect defects. However, only the number of detected defects in each phase can be known with any accuracy prior to project completion. The number of defects inserted in each phase cannot be known until all defects have been found. Confidence in knowing that approximate number comes only after the system has been fielded. Consequently, this approach focuses on defects detected.

The techniques presented here depend on two key assumptions:

- Size is the easily quantifiable software attribute that is most closely associated with the number of defects. The basic test of the effectiveness of complexity models and other indicators of defect-proneness is to ask, "Does this model show a significantly higher correlation with defects than just size (e.g., lines of code) alone?" [5].
- Defect insertion and detection rates tend to remain relatively constant as long as the project's software processes remain stable. While the rates are not exactly constant, they perform within a recognized range.

The first assumption appears to be inherent to the nature of software. CMM Level 4 organizations actively work to make the second assumption come true. That is, they are acting to bring their processes under control.

An Empirical Model

The simplest approach to generating a defect profile for intended projects within the organization is to collect actual data about the insertion and detection rates in each life-cycle phase. This can be accomplished in the following four steps:

- First, historical data are collected. Table 1 shows a simple spreadsheet used to tabulate defect discovery and detection data using example data. In addition, the size of the project from which the defect data is collected must be known. The size measure must be applied consistently, but this approach does not depend on using any specific measure. Lines of code, function points, number of classes, etc., may be used as appropriate. (The data in Table 1 are simulated, not real.)
- Second, an initial profile of the number of defects found in each phase is gener-

Table 1: *Example of Empirical Defect Profile (Simulated Data)*

	Phase Inserted					
Phase Detected	Analysis	Design	Code	Developer Test	System Test	**Total**
Analysis	0					0
Design	50	200				250
Code	50	100	300			450
Developer Test	25	50	150	0		225
System Test	18	38	113	0	0	169
Operation	7	12	37	0	0	56
Total	150	400	600	0	0	1,150

® Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.
℠ Capability Maturity Model Integration and CMMI are service marks of Carnegie Mellon University.

ated as shown in Figure 1. The bars in that figure represent the totals in the last column of Table 1.

- Third, this initial profile is scaled to account for differences between the size of the project(s) from which the profile was developed and the size of the project to which it is applied. This is accomplished by multiplying by the ratio of the project sizes. For example, if the defect profile in Figure 1 were to be used to develop a defect profile for a project twice the size of the project providing the data that went into Figure 1, then the bars of the profile representing the new project would be twice the size of those in Figure 1.
- Fourth, the scaled defect profile is adjusted further to reflect the planned performance of the project. For example, if the project plan called for the automatic generation of code from design instead of hand coding as previously done, then the number of defects inserted in the implementation phase would be adjusted downward to reflect this change in the coding process. Also, changes in the project's process may be induced in order to reach a specified target in terms of delivered quality if previous performance did not yield the required level of quality. The target might be specified as a result of a customer requirement or an organizational goal.

Actual defect counts can then be compared with this final plan (defect profile) as the project progresses. Suggestions for this activity are provided in a later section of this article. Note that the defect profile does not address defect status (i.e., *open* vs. *closed* problems/defects). All detected defects, regardless of whether or not they ever get resolved, are included in the defect counts.

Figure 2 shows an example of a defect profile developed empirically [6] for an actual military project. This figure shows the predicted number of defects to be injected and detected in each phase, based on previous projects. However, only actual counts are shown for the number of defects detected, because the actual number injected cannot be determined with any confidence until after software delivery.

The project in Figure 2 was about two-thirds of the way through software integration at the time data were reported. Two-thirds of the predicted number of defects had been found in software integration. The project's quality performance was tracking the plan. This illustrates that the real value of the defect profile lies in its ability to make quality visible during development, not as a post-mortem analysis technique.

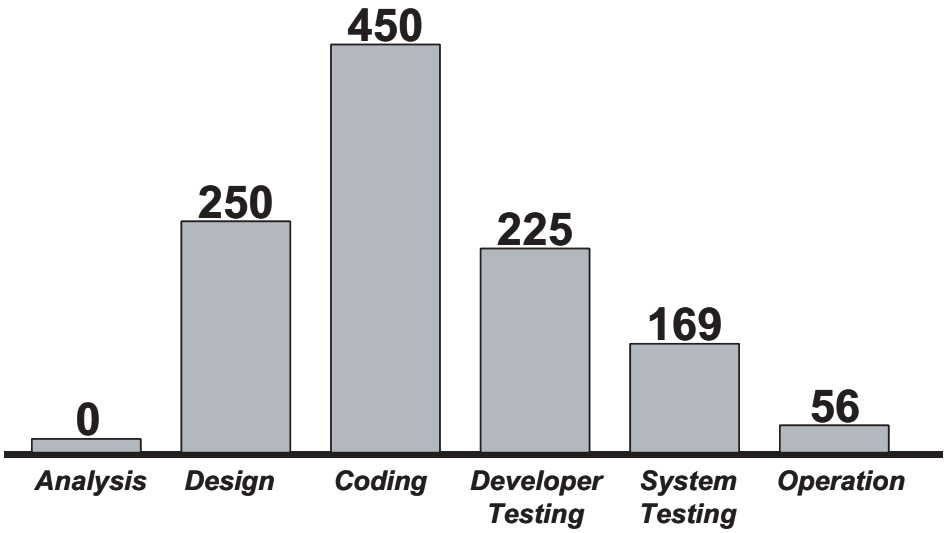


Figure 1: *Example of Defect Profile With Data From Table 1*

The project in Figure 2 actually was completed after this graph was prepared. The planned and actual defect levels never differed by more than 10 percent. The project team handed their product over to the customer with a high degree of confidence that it met the targeted level of quality.

An Analytical Model

Defect profiles may also be generated analytically. Many early studies of defect occurrence suggest that they followed a Rayleigh dispersion curve, roughly proportional to project staffing. The underlying assumption is that the more effort expended, the more mistakes that are made and found.

Gaffney [7] developed one such model:

$$Vt = E\,(1 - \exp(-B(t^{**}2)))$$

Where:

Vt = Number of defects discovered by time *t*.

E = Total number of defects inserted.

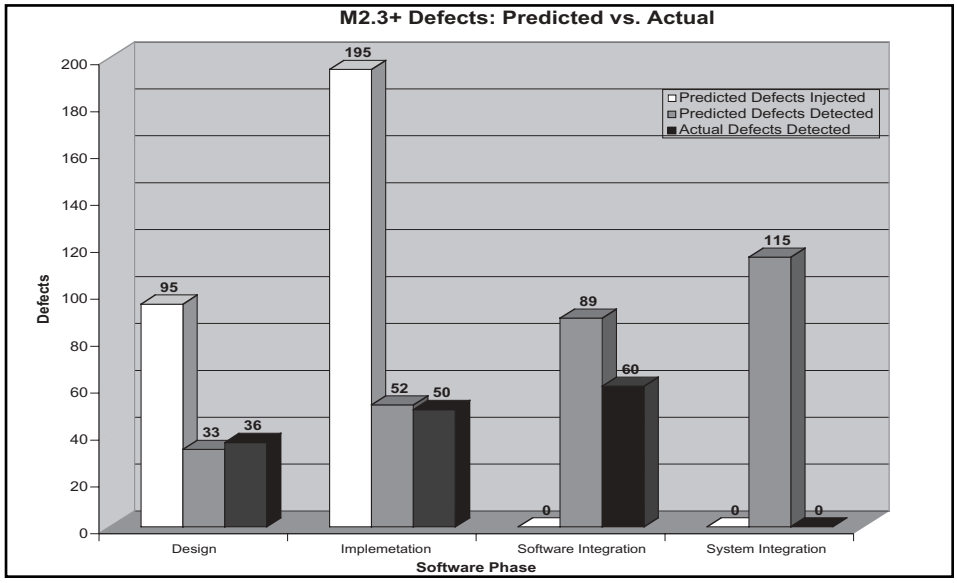
B = Location parameter for peak.

The time periods *t* can be assumed to be equal to life-cycle phase transition boundaries in order to apply the model to project phases rather than elapsed time. The location parameter *B* fixes the time of the maximum (or peak) distribution. For example, *B=1* means that the peak occurs at *t*=1.

The analytical approach involves applying regression analysis to actual phase-by-phase defect data to determine the values of *B* and *E* that produce a curve most like the input data. Many Software Productivity Consortium member companies use our proprietary software, SWEEP [8] (based on the Gaffney model), to perform this analysis, but it can easily be implemented in Microsoft Excel.

The effectiveness of the analytical approach depends on the satisfaction of additional assumptions, including the following:

Figure 2: *Example of Empirical Defect Profile (Actual Project Data)*



- Unimodal staffing profile.
- Life-cycle phases of similar (not exactly equal) duration (not effort).
- Reasonably complete defect reporting.
- Using only observable/operational defects.

To the extent that these assumptions are satisfied, this model gives better results. Analytical models such as this are useful when the organization lacks complete life-cycle defect data or desires to smooth existing data to provide an initial solution for new projects without prior historical data. The defect profile obtained from the actual data can be easily adjusted to fit projects with different numbers of life-cycle phases and processes by selecting appropriate values of *E* and *B*.

Figure 3 provides an example of a defect profile for another actual military project generated by SWEEP. The light bars in Figure 3 represent the expected number of defects for each phase, based on the model. For this specific project, the actual number of defects discovered is substantially lower than planned during design. Consequently, additional emphasis was placed on performing rigorous inspections during code, with the result that more defects than anticipated were captured during code, putting the project back on track to deliver a quality project as shown at post release.

A detailed discussion and analysis of applying the Gaffney model to a military project using SWEEP can be found in [9].

Interpreting Differences

During project execution, planned defect levels are compared to actual defect levels. Typically, this occurs at major phase transitions (milestones). However, if a phase extends beyond six months, then consider inserting additional checkpoints during the phase (as in the example in Figure 1 where analyses were conducted at the completion of each one-third of integration testing). Since real performance never exactly matches the plan, the differences must be investigated. This involves three steps:

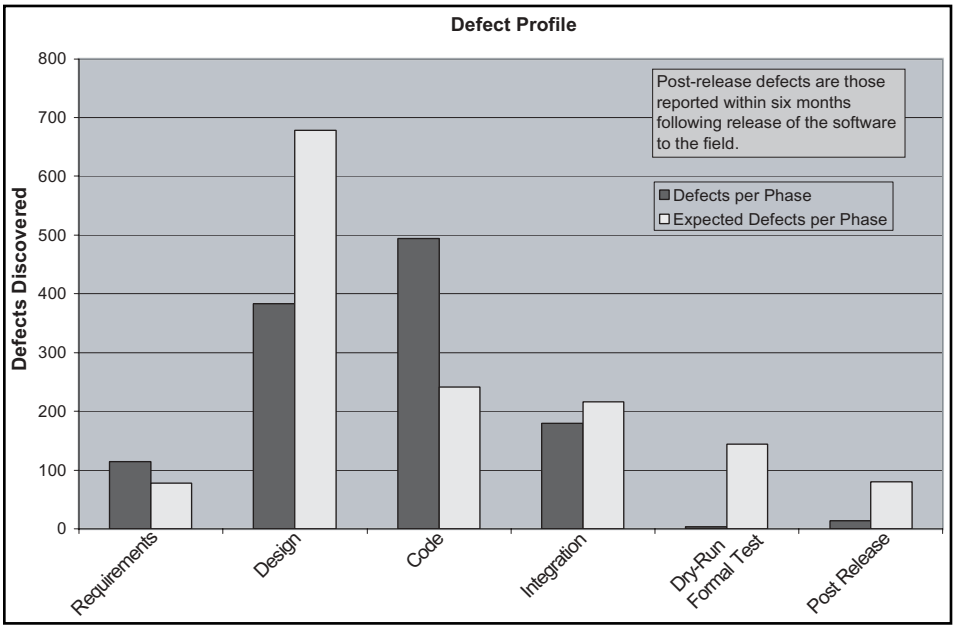
- Determine if the differences are significant and/or substantive. This might be accomplished by seeking visually large differences, establishing thresholds based on experience, or applying statistical tests such as the Chi-Square [10].
- Determine the underlying cause of the difference. This may require an examination of other types and sources of data such as process audit results as well as effort and schedule data. Many techniques have been developed for causal analysis (e.g., [11]), but they fall beyond the scope of this article.
- Take appropriate action. This includes corrective actions to address problems identified in the preceding step, as well as updates to the defect profile to reflect anticipated future performance.

Differences between planned and actual defect levels do not always represent quality problems. Potential explanations of departures from the plan include the following:

- Bad initial plan (assumptions not satisfied, or incomplete or inappropriate data).
- Wrong software size (more or less than the initial estimate).
- Change in process performance (better or worse than planned).
- Greater or lesser software complexity than initially assumed.
- Inspection and/or test coverage not as complete as planned.

Analyzing departures from the defect profile early in the life cycle provides feedback for our understanding of the size and complexity of the software engineering task while there is still time to react.

Figure 3: *Example of Analytical Defect Profile (Actual Project Data)*



Summary

Relatively simple models of software quality based on defect profiles are becoming increasingly popular in the software industry as organizations mature. These models establish a *quality budget* that helps to make trade offs among cost, schedule, and quality visible and reasoned, rather than choices made by default. Defect profiles present quality performance to the project manager in a form that he or she understands. Thus, the consequences of a decision such as "reducing inspection and testing effort to accelerate progress" can be predicted. Unintended departures from planned quality activities can be detected and addressed.

Moreover, the ability to model quality across the project life cycle is a necessary prerequisite to implementing design for Six Sigma techniques [3] in software development. Achieving Six Sigma requires measuring and managing quality at each software production step, not just during the final testing stages prior to delivery.

Defect models can become very rich. The concept of orthogonal defect classification [12], for example, involves developing separate profiles for each of many different defect types. These defect classifications facilitate the causal analysis process when potential problems are identified.

This article discussed two very simple approaches to building and using defect profiles. These techniques make quality visible so that it can be managed.◆

References

1. Paulk, Mark, et al. Capability Maturity Model: Guidelines for Improving the Software Process. Boston: Addison-Wesley, June 1995.
2. Software Engineering Institute. Capability Maturity Model® Integrated℠. Pittsburgh: SEI.
3. Harry, Mikel J., and Richard Schroeder. Six Sigma: The Breakthrough Management Strategy Revolutionizing The World's Top Corporations. New York: Doubleday, Dec. 1999.
4. ISO/IEC Standard 9126. "Information Technology – Software Quality, Part 1." 1995.
5. Card, David, and William Agresti.

"Resolving the Software Science Anomaly." Journal of Systems and Software Vol. 7 (1990): 29-35.
6. Card, David. "Quantitatively Managing the Object-Oriented Design Process." Canadian National Research Council Conference on Quality Assurance of Object-Oriented Software. Feb. 2000.
7. Gaffney, John. "Some Models for Software Defect Analysis." Lockheed Martin Software Engineering Workshop, Gaithersburg, MD, Nov. 1996.
8. Software Productivity Consortium. SWEEP Users Guide. SPC-98030-MC, 1997.
9. Harbaugh, Sam. "Crusader Software Quality Assurance Process Improvement." Technical Report. Integrated Software, Inc., 2002.
10. Hays, William, and Robert Walker. Statistics: Probability, Inference, and Decision. Austin, TX: Holt, Rinehart, and Winston, 1970.
11. Card, David. "Learning From Our Mistakes With Defect Causal Analysis." IEEE Software Jan. 1998.
12. Chillarge, R., et al. "Orthogonal Defect Classification." IEEE Transactions on Software Engineering Nov. 1992.

Note

1. An earlier version of this article was published in the proceedings of the Institute of Electrical and Electronics Engineers' Computer Software and Applications Conference, Aug. 2002.

About the Author

David N. Card is a fellow of the Software Productivity Consortium where he provides technical leadership in software measurement and process improvement. During 15 years at Computer Sciences Corporation, Card spent six years as the director of Software Process and Measurement, one year as a resident affiliate at the Software Engineering Institute, and seven years with the research team supporting the NASA Software Engineering Laboratory. Card is editor-in-chief of the *Journal of Systems and Software*. He is the author of "Measuring Software Design Quality," co-author of "Practical Software Measurement," and co-editor of ISO/IEC standard 15939:2002 "Software Measurement Process." Card is a senior member of the American Society for Quality.

Software Productivity Consortium
2214 Rock Hill Road
Herndon, VA 20170
Phone: (703) 742-7199
Fax: (703) 742-7200
E-mail: card@software.org

WEB SITES

Software Quality HotList

www.soft.com/Institute/HotList

The Software Research Institute maintains a list of links to selected organizations and institutions that support the software quality and software testing area. Organizations and other references are classified by type, by geographic area, and then in alphabetic order within each geographic area. The institute's aim is to bring to one location a complete list of technical, organizational, and related resources.

The Quality Assurance Institute

www.qaiusa.com

The Quality Assurance Institute (QAI) is exclusively dedicated to partnering with the enterprise-wide information quality profession. QAI is an international organization consisting of member companies in search of effective methods for detection-software quality control and prevention-software quality assurance. QAI provides consulting, education services, and assessments.

Software Technology Support Center

www.stsc.hill.af.mil

The Software Technology Support Center is an Air Force organization established to help other U.S. government organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

COMING EVENTS

March 24-28
International Symposium on Integrated Network Management
Colorado Springs, CO
www.im2003.org

March 31-April 2
Association for Configuration and Data Management's Annual Technical and Training Conference
San Diego, CA
www.acdm.org/main.htm

April 1-2
SecurE-Biz Summit
Arlington, VA
www.SecurE-Biz.net

April 8-10
FOSE 2003
(Federal Office Systems Exposition)
Washington, D.C.
www.fose.com

April 28-May 1
Software Technology Conference 2003

Salt Lake City, UT
www.stc-online.org

May 3-10
International Conference on Software Engineering
Portland, OR
www.icse-conferences.org/2003

May 12-16
STAREAST '03
Orlando, FL
www.sqe.com/stareast/

June 2-6
Applications of Software Measurement
San Jose, CA
www.sqe.com/asm

August 19-22
Software Test Automation Fall '03
Boston, MA
www.sqe.com/testautomation/

Lean Six Sigma: How Does It Affect the Government?

Dr. Kenneth D. Shere
The Aerospace Corporation

Lean Six Sigma (LSS) is a combination of historical methods for process improvement that focuses on the bottom line and critical-to-customer requirements. This method differs from previous process improvement approaches because it uses established engineering principles and is based on institutionalization of the approach and independent validation of claims of success. LSS has been highly successful in industry, but the government has largely ignored it. This article provides an introduction to LSS and describes how the government can benefit from using "LSS thinking" in system acquisition.

Lean Six Sigma (LSS) is the culmination of a variety of process improvement methods. These methods began in the 1920s with the development of time and motion studies, and the principles of statistical quality control. Thirty years later in the early 1950s, W. Edwards Deming and Bonnie Small developed the foundations of modern process improvement methods. Deming developed Total Quality Management (TQM). Small made the analyses of statistical quality control accessible to people who were not professional statisticians and mathematicians through her publication of "The Western Electric Rules" [1].

Prior to the development of LSS, process improvement methods were narrowly focused. They did not address the bottom line in terms of *what is critical to the customer* and *the cost of poor quality*.

Lean manufacturing focuses on eliminating *nonvalue-added* and *unnecessary* tasks. Tasks are value-added when the customer is willing to pay for them. Some tasks like invoicing are nonvalue-added, but are necessary for business operations. The lean methodology is bottom-line focused but does not address quality *per se*. Motorola [2] developed Six Sigma to drive defects to zero, but did not explicitly address the elimination of unnecessary tasks.

LSS is an approach that combines lean manufacturing and Six Sigma from a global perspective that takes both suppliers and customers into account. This approach tells us how to improve our processes in a way that considers both the costs of poor quality and issues critical to customer requirements. In addition to manufacturing processes, LSS has been very successfully used in transactional and service industries. It also directly applies to software processes, but few organizations have applied it.

The companies that are the strongest proponents of LSS include General Electric Co., Sony Corporation, Honeywell, TRW Inc., Bombardier, Johnson and Johnson, The Dow Chemical Company, Exxon Mobil Corp., J.P. Morgan Chase & Co., Citibank, GMAC Mortgage Corporation, and John Deere. In annual meetings and letters to shareholders, these companies have credited LSS with saving billions of dollars in operational expenses.

Successful LSS application requires committed leadership, education, and institutionalization. Regardless of future names and improvements, LSS as a concept will continue. The approach is flexible in the sense that the methodology is not intended to be static. LSS applies its basics to itself, i.e., just as LSS is used to continuously improve other processes, it should be used to continuously improve the improvement process.

> *"This approach [Lean Six Sigma] tells us how to improve our processes in a way that considers both the costs of poor quality and issues critical to customer requirements."*

Among process improvement approaches like TQM, business process reengineering and the Capability Maturity Model®, only LSS requires each of the following activities: (1) focusing on what is critical to the customer, (2) emphasizing the bottom line, (3) validating any claims of success, and (4) institutionalizing the process through extensive training programs and certification of expertise.

This methodology could be important to the military for several reasons:

- LSS has been proven by industry to be highly successful.
- Major prime contractors have implemented LSS, including The Boeing Company, Raytheon Company, Lockheed Martin Corporation, TRW, Honeywell, and Northrop Grumman Corporation.
- LSS can help the military operate more efficiently.
- LSS thinking can be applied to acquisitions and software intensive systems.

This article provides some answers to the following questions:

- What is LSS?
- Why should the government care about LSS?
- How can LSS be applied to acquisition?

What Is LSS?

Six Sigma was developed by Motorola Inc. in the mid-1980s to control variability in processes. Simply stated, Motorola concluded that they could not compete with the Japanese using their current concept of quality. The cost of poor quality was too high. They developed Six Sigma to produce essentially zero defects in their products.

Lean manufacturing inspects the process by analyzing each task or activity to determine whether it is value-added, is not value-added but necessary, or is not value-added. A value-added activity is something for which the customer is willing to pay. An example of a value-added activity is the maintenance of a satellite operations center. If a contractor was maintaining this center, then an example of a nonvalue-added but necessary activity is an invoice payment. Activities that neither add value nor are necessary should be eliminated.

When a major program review is held, like the critical design review for the software of a major system, it is common for the review to be attended by 150 high-priced people and to last for a week. These reviews are frequently dog-and-pony shows in which no really critical review takes place. A typical cost for this review is more than $8/second (more

than $1 million for the week). Does this review add value? How should it be changed to add value? LSS thinking addresses these questions.

LSS is a defined approach that synthesizes the use of established tools and methods. Its methods are generally divided into two approaches. One approach is called *design for Six Sigma*. It is generally used when designing new systems or processes.

The other approach, used for process improvement, is called the *define-measure-analyze-improve-control* approach, which represents five phases. Some organizations (and I agree with them) use six phases. The difference between these representations is that the second approach divides the define phase into vision and define. These six phases of LSS are described below.

- **Vision.** This phase is used to identify critical-to-customer factors, teams, and key stakeholders; to describe the business impact; and to plan the process improvement project.
- **Define.** This phase focuses on defining the as-is process. Frequently, processes are understood by experienced personnel but are not actually written down. Simply gathering a group of key people in a room and asking them to define a process often improves it. Sometimes the improvements are significant, and the team decides that it is good enough – no further work is necessary.
- **Measure.** The purpose of this phase is to measure the existing process. Without these measurements, it is impossible to determine how much a process is improved or to validate savings. This phase is critical to future analyses and suggested process improvements.
- **Analyze.** During this phase, the causes of poor quality are determined and analyzed. Each step of the process is assessed to determine waste from a lean perspective. Problems are determined from historical data and employee knowledge. Fishbone charts, also called cause-and-effect charts, are used to identify the most likely causes of the defects. The process is usually simulated to determine bottlenecks and resource utilization, and the cost of defects. These analyses form the basis for design of experiments, regression analysis, and other techniques used to evaluate potential improvements in the next phase.
- **Improve.** The focus of this phase is to determine process improvements. Processes are assessed from the perspectives of whether (1) each task adds value to the product or service, (2) there is a more cost-effective way of performing the process, and (3) the process meets or accounts for requirements critical to the customer. Typically, the process is modeled and simulated. New ideas are tried out in simulation before they are implemented. Sometimes it is necessary to perform a design-of-experiments analysis to determine ways to improve a process. This allows the analyst to determine the value of adding people or resources to a given task or taking them away from another task. It also allows the analyst to look at fundamental changes to the process. These analyses are conducted from a bottom-line perspective.
- **Control.** During this phase, the improved process is implemented in a controlled manner. Data are taken to verify that the proposed improvement (previously validated through simulations) is real. The financial member of the team serves as an independent auditor and validates the savings. Frequently, the process is initially implemented as a prototype before full implementation.

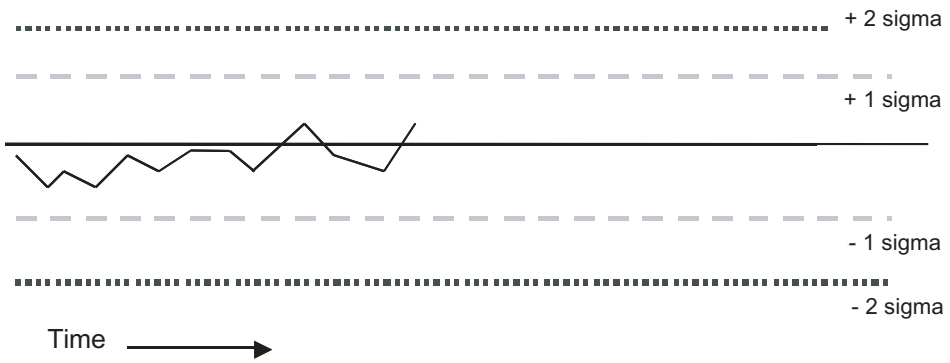


Figure 1: *Sample Telemetry Run-Chart*

"Lean manufacturing inspects the process by analyzing each task or activity to determine whether it is value-added, is not value-added but necessary, or is not value-added."

Each of these phases uses a defined methodology. Training to become an expert in LSS takes several weeks spread over a five-month period. Between the classes, students work on an actual project, receiving consulting advice from the trainer as necessary. When training is completed, the student will have implemented a successful LSS project. After completing a second project in which the student is the leader, and taking an exam, the student is certified as a LSS expert, or equivalent to a *black belt* in the field. There are lesser levels of training for people who help on LSS projects but are not the leaders. LSS organizations provide training internally, but consulting companies are generally used in the early stages of implementing LSS.

This article has talked about LSS as a methodology, but it is more than that. It is a way of thinking that is illustrated in the following way. Consider the telemetry run chart shown in Figure 1. A satellite operator looking at this chart in real time thinks that everything looks pretty good. In fact, it looks great – the process is running well within its control limits.

If we apply LSS thinking to this chart, we begin to analyze the process statistically. Applying well-known run-chart analysis techniques [1] (which could be automated), we see that a statistically significant event has already occurred and another might occur soon. These events are identified in Figure 2 (see page 10).

The fact that a statistically significant event has occurred does not necessarily mean that something bad has happened. It does mean that there is an anomaly that needs explanation. This analysis would trigger involvement by satellite engineers to resolve the anomaly. If the event is caused by something bad, the engineers may be able to resolve it before additional damage is done to the satellite. This example shows how LSS thinking can lead to early detection of anomalies.

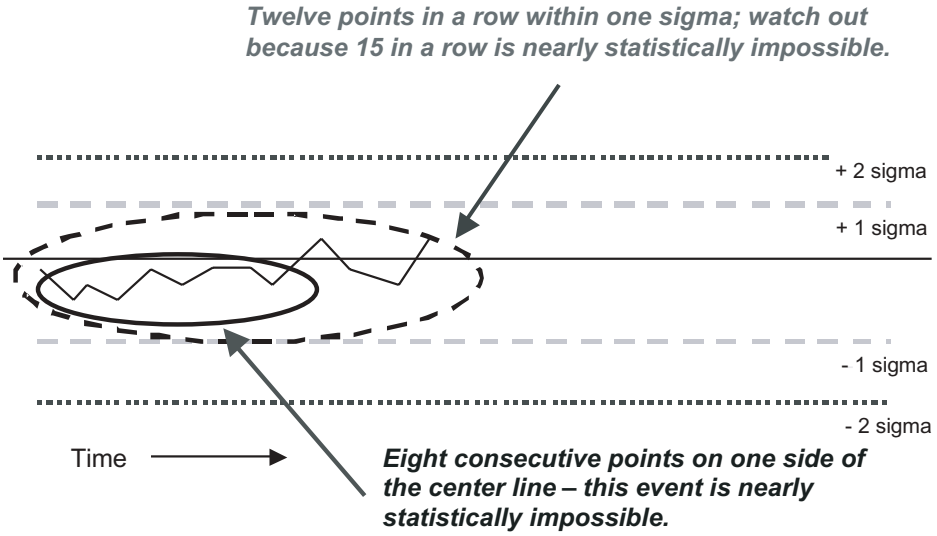


Figure 2: *Run-Chart With Statistical Analyses Superimposed*

Why Should the Government Care About LSS?

LSS is a best practice for process improvement. It applies to all processes – manufacturing, software, operational, transactional, and service processes. All the major space contractors use LSS, including Lockheed Martin, Raytheon, Boeing, TRW, Northrop Grumman, and Honeywell. Furthermore, the presidents of Lockheed Martin, Honeywell, Raytheon, and TRW are among the leading advocates for LSS in the defense industry.

Many of these companies actively promote its use. At Lockheed Martin, their LM21 Operating Excellence program is based on LSS. The Raytheon Learning Institute is offering LSS services to external companies as well as for training people throughout Raytheon. Honeywell has integrated LSS into its software processes. TRW has a major training program underway that will touch every TRW employee. To answer a question with a question: If the major prime contractors believe in LSS and are applying it in both their government and commercial business, why has the government largely ignored it?

Lt. Gen. Brian A. Arnold, commander, Space and Missile Systems Center, Los Angeles Air Force Base, was recently quoted as saying, "I will tell you that in virtually every one of our major programs we are out of control on cost and schedule" [2]. LSS is designed for process improvement, but its principles can help maintain both cost and schedule control. LSS is based on two perspectives: requirements that are critical to the customer, and satisfying these requirements at the lowest possible cost.

The first perspective limits requirements creep – a major driver of both cost and schedule growth. Before imposing a new requirement, either in the specification development phase or in the system development phase, the program manager should ask, "Is this requirement really critical to the customer?" Another way of asking this is to determine what the customer is willing to pay for the additional requirement. The *nice-to-have* requirements frequently fall by the wayside under this type of scrutiny.

The second perspective is more complicated because its answer depends on a total system/result perspective. This perspective forces us to think about the cost of poor quality. Providing requirements at the lowest possible cost is a driver for using a defined systems engineering methodology. A defined methodology with good documentation and built-in quality significantly lowers the cost of operations and maintenance. This claim is not a *logical* but *subjective conclusion*. Organizations that have, for example, implemented the Software Engineering Institute's Capability Maturity Model® (CMM®) to Level 4 or 5 have the data to prove it. LSS companies also have the data to prove it.

Knowledge of LSS is critical to government agencies. This knowledge could be applied to do the following:

- Help operations groups improve their processes.
- Help the transactional groups (e.g., finance, contracts, and human resources) improve their processes.
- Specify contractor incentives to be included in awards.
- Understand what performance information to request from contractors.
- Evaluate contractor proposals.
- Assure that contractors apply best practices to their customers' programs.

A further discussion of how to apply LSS to acquisition follows in this article.

How does LSS apply to the military? The Advanced Extremely High Frequency (AEHF) satellite communications system will cost $3.19 billion to produce two satellites, each with 10 years of operational life [3]. Assume that there will be no cost overrun, that each satellite will be successfully launched with full operational capability for its entire life, and that there are *no* operations cost. With these assumptions, the straight-line amortized cost of this satellite system will be about $437,000 per day per satellite throughout its operational life. A 95 percent uptime means that the downtime costs $160 million (or one year of operation). A 99 percent uptime means that the downtime costs $32 million (or 2.4 months of operation). LSS, or 99.9997 percent uptime, means that the downtime costs $6.57 (or 1.3 seconds of operation). The impact of downtime on military operations is immeasurable.

For some organizations, applying LSS seems pretty clear. Any group operating and maintaining an information technology enterprise cannot survive without processes. These processes include helpdesk processes, logistics processes, property management processes, and so on. These activities sound like simple, mundane stuff until you consider their size. The information technology enterprise of one of our customers requires more than 1,200 people for operations and maintenance. Other potential applications are the human resources and financial processes. Note that each of the processes mentioned use a substantial amount of software.

How Does LSS Apply to System Acquisition?

Stating that LSS applies directly to system acquisition because it is a process is correct, but it is a cop-out. The focus of this section is how to use aspects of LSS during pre-proposal activities, proposal evaluation, and program evaluation. The government cannot require companies to use LSS, just like they cannot require companies to use the CMM. However, these items can be used as factors during an evaluation.

Pre-Proposal Activities

The greatest value of LSS thinking is obtained during the pre-proposal stage. This is the time when program managers can think proactively. During this phase, key decisions are made involving the structure of the future contract, the meas-

ures that will be used to evaluate contractor performance, and the criteria for evaluating bidders.

Every prime contractor has a set of corporate engineering and management processes that reflect best practices. Most of these contractors have policies about implementing LSS. Structure your request for information (RFI) in a way that gets the contract personnel to use these processes. Some of the questions that could be included in an RFI include the following:

- What are your corporate processes and policies related to LSS or process improvement?
- What corporate processes will be used on this contract?
- How will the performance of these processes be measured?
- How do you improve your processes during the performance of a contract?
- How do you assure that these corporate processes are used and improved?
- Are the key people who will be working on this effort trained and certified in your corporate processes?
- How will you measure the improvement?
- What is your process and criteria for handling changes to requirements?
- What is your procedure for determining the cost of poor quality?
- What recommendations do you have for structuring an incentive clause?
- What recommendations do you have for measuring contractor performance?

Proposal Evaluation

Determine the extent to which the contractor is using LSS thinking. For example, in answering the question on requirements changes, does the contractor address the issue of whether a change is critical to the customer? If so, then the cost and schedule impact of every change needs to be determined. The contractor then needs to meet with the government and ask whether the government is willing to pay for the change. The cost for this change is more than simply money. From an oversight perspective, additional cost means overrun. It usually does not matter whether the cost is due to a new, critical requirement. The contractor needs to determine the likely savings resulting from future process improvements. These cost savings might offset the budgetary impact of a new critical requirement.

Government program offices need to understand that they need to work with the contractor on the issue of requirements criticality. Many contractors think that any change requested by the contracting officer's technical representative is critical – the acquisition office is the customer. From a government perspective, there are multiple customers, including the users (frequently operational forces), Congress, and other agencies.

The contractor needs to be willing to challenge whether a new requirement is really critical. Current contracting approaches force contractors to bid unreasonably low under the assumption that the deficit will be eliminated through engineering change proposals. That is a losing approach because overruns are assured. Think about the win-win approach used in LSS. Use incentive fees for contractors to benefit from reducing total cost to the government. Fund new critical requirements to the maximum extent possible from the government's share of the savings resulting from process improvement.

Looking at the cost of poor quality helps program managers address difficult questions. Budgetary pressures and external policies frequently drive program managers to make decisions that they know are *penny-wise but pound-foolish*. For example, by separating development costs from operations and maintenance costs, program managers are driven to make decisions favoring reduced development cost even though a severe impact might be realized in operations. A detailed cost of poor quality analysis may enable program managers to justify different decisions. These types of decisions significantly impact the system that will be built.

Program Evaluation

If LSS thinking was used during the preproposal and proposal evaluation phases, its use during program evaluation is mundane. The metrics are defined, so the government can use them to evaluate contractor performance. This approach seems straightforward and simple, but its proper execution requires program managers to have some training in LSS; the preferable level is equivalent to a *green belt*. This training typically involves one week of classwork followed by working on a LSS project in a support role.

Obtaining this week of training is generally not a problem, but many program managers will not have the time to actually work a process improvement effort using LSS. I recommend that program managers take a one-week course on LSS. This training will provide a more thorough understanding of the process, and they will be able to ask appropriate questions of the contractor to verify that the methodology is actually being used.◆

References

1. Small, Bonnie. The Western Electric Rules. Code 700-444. AT&T Lucent Technologies, 1984. Report of Statistical Quality Control Handbook. Western Electric Co., 1956.
2. Tuttle, Rich. "Arnold: Most Big Air Force Space Programs Facing Cost, Schedule Difficulty." The Aerospace Daily 200.43 (30 Nov. 2001): 1.
3. Jonson, Nick. "Lockheed Martin Awarded $498 Million Contract for AEHF System." The Aerospace Daily 202.40 (24 May 2002): 4.

Additional Reading

1. Harry, Mikel J., and Richard Schroeder. Six Sigma: The Breakthrough Management Strategy Revolutionizing the World's Top Corporations. New York: Doubleday, Dec. 1999.
2. Breyfogle, Forrest. Implementing Six Sigma: Smarter Solutions Using Statistical Methods. Hoboken, NJ: Wiley-Interscience, June 1999.
3. Pande, Peter S., Robert P. Neuman, and Roland P. Cavanagh, The Six Sigma Way: How GE, Motorola, and Other Top Companies Are Honing Their Performance. Columbus, OH: McGraw-Hill, Apr. 2000.

About the Author

Kenneth D. Shere, Ph.D., is a senior engineering specialist at The Aerospace Corporation where he provides systems and software engineering, acquisition, and strategic leadership support to various government organizations. He is certified as a Lean Six Sigma green belt and a software capability evaluator. Shere has published 18 articles and two books. He has a bachelor's of science degree in aeronautical and astronautical engineering, a master's of science degree in mathematics and a doctorate in applied mathematics, all from the University of Illinois.

The Aerospace Corporation
15049 Conference Center Drive
Chantilly, VA 20151
Phone: (703) 633-5331
Fax: (703) 633-5006
E-mail: kenneth.d.shere@aero.org

What Is Requirements-Based Testing?

Gary E. Mogyorodi
Bloodworth Integrated Technology, Inc.

This article provides an overview of the requirements-based testing (RBT) process. RBT is comprised of two phases: ambiguity reviews and cause-effect graphing. An ambiguity review is a technique for identifying ambiguities in functional [1] requirements to improve the quality of those requirements. Cause-effect graphing is a test-case design technique that derives the minimum number of test cases to cover 100 percent of the functional requirements. The intended audience for this article is project managers, development managers, developers, test managers, and test practitioners who are interested in understanding RBT and how it can be applied to their organization.

The requirements-based testing (RBT) process is comprised of two phases: ambiguity reviews and cause-effect graphing. An ambiguity review is a technique for identifying ambiguities in functional[1] requirements to improve the quality of those requirements. Cause-effect graphing is a test-case design technique that derives the minimum number of test cases to cover 100 percent of the functional requirements.

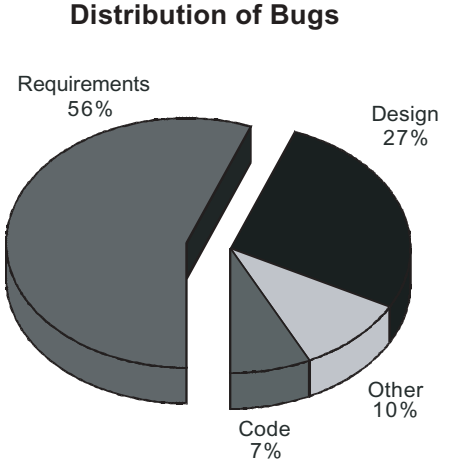
Testing can be divided into the following seven activities:

1. **Define Test Completion Criteria.** The test effort has specific, quantifiable goals. Testing is completed only when the goals have been reached (e.g., testing is complete when the tests that address 100 percent functional coverage of the system all have executed successfully).
2. **Design Test Cases.** Logical test cases are defined by four characteristics: the initial state of the system prior to executing the test, the data, the inputs, and the expected results.
3. **Build Test Cases.** There are two parts needed to build test cases from logical test cases: creating the necessary data, and building the components to support testing (e.g., build the navigation to get to the portion of the program being tested).
4. **Execute Tests.** Execute the test-case steps against the system being tested and document the results.
5. **Verify Test Results.** Testers are responsible for verifying two different types of test results: Are the results as expected? Do the test cases meet the test completion criteria?
6. **Verify Test Coverage.** Track the amount of functional coverage achieved by the successful execution of each test.
7. **Manage the Test Library.** The test manager maintains the relationships between the test cases and the programs being tested. The test manager keeps track of what tests have or have not been executed, and whether the executed tests have passed or failed.

Activities one, two, and six are addressed by RBT. The remaining four activities are addressed by test management tools that track the status of test executions.

The RBT process stabilizes the application interface definition early because the requirements for the user interface become well defined and are written in an unambiguous and testable manner. This allows the use of capture/playback tools sooner in the software development life cycle.

Figure 1: *Distribution of Bugs*



Relative Cost to Fix an Error

The cost of fixing an error is lowest in the first phase of software development (i.e., requirements). This is because there are very few deliverables at the beginning of a project to correct if an error is found. As the project moves into subsequent phases of software development, the cost of fixing an error rises dramatically since there are more deliverables affected by the correction of each error. At the requirements phase the cost ratio to fix errors is one to one; at coding it is 10 to one; at production it is from 40 to 1,000 to one.

A study by James Martin showed that the root cause of 56 percent of all bugs identified in projects is errors introduced in the requirements phase. Of the bugs rooted in requirements, roughly half were due to poorly written, ambiguous, unclear, and incorrect requirements. The remaining half was due to requirements that were completely omitted (see Figure 1).

Why Good Requirements Are Critical

A study by the Standish Group in 2000 showed that American companies spent $84 billion for cancelled software projects. Another $192 billion was spent on software projects that significantly exceeded their time and budget estimates. The Standish Group and other studies show there are three top reasons why software projects fail:

- Requirements and specifications are incomplete.
- Requirements and specifications change too often.
- There is a lack of user input (to requirements).

The RBT process addresses each of these issues:

- It begins at the first phase of software development where the correction of errors is the least costly.
- It begins at the requirements phase where the largest portion of bugs have their root cause.
- It addresses improving the quality of requirements: Inadequate requirements often are the reason for failing projects.

A Good Test Process

The characteristics of a good test process are as follows:

- **Testing must be timely**. Testing begins when requirements are first drafted; it must be integrated throughout the software development life cycle. In this way, testing is not perceived as a bottleneck operation. Test early, test often.
- **Testing must be effective.** The approach to test-case design must have rigor to it. Testing should not rely on individual skills and experiences. Instead, it should be based on a repeat-

able test process that produces the same test cases for a given situation, regardless of the tester involved. The test-case design approach must provide high functional coverage of the requirements.

- **Testing must be efficient.** Testing activities must be heavily automated to allow them to be executed quickly. The test-case design approach should produce the minimum number of test cases to reduce the amount of time needed to execute tests, and to reduce the amount of time needed to manage the tests.
- **Testing must be manageable.** The test process must provide sufficient metrics to quantitatively identify the status of testing at any time. The results of the test effort must be predictable (i.e., the outcome each time a test is successfully executed must be the same).

Standard Software Development Life Cycle

There are many software development methodologies. Each has its own characteristics and approaches, but most software development methodologies share the following six aspects:

- **Requirements.** There is a description of what has to be delivered.
- **Design.** There is a description of how the requirements will be delivered.
- **Code.** The system is constructed from the requirements and the design.
- **Test.** The behavior of the code is compared to the expected behavior described by the requirements.
- **Write user manuals/write training materials.** Documentation is created to support the delivered system.
- **International translations.** Code is often executed in different countries with different languages; the initial system must be translated into the native language of the target country.

In many software development methodologies, testing does not begin until after code is constructed. If a defect is found after coding, there is a good deal of scrap and rework to correct the code, and possibly the design, test cases, and requirements as well. Defects must be tested out of the system rather than being avoided in the first place. Testing often is a bottleneck activity. See Figure 2 for a graphical representation of a standard development life cycle.

Life Cycle With Testable Requirements

In a software development life cycle with testable requirements and integrated testing, the RBT process is integrated throughout the entire software development life cycle. As soon as requirements are complete, they are tested. As soon as the design is complete, the requirements are walked through the design to ensure that they can be met by the design. As soon as the code is constructed and reviewed, it is tested as usual. But because testing begins at the requirements phase, many defects are avoided instead of being tested out of the code.

This is a less costly and more timely approach. User manuals and training materials can be developed sooner. The entire software development life cycle is compressed. Testing is performed in parallel with development instead of all at the end, so testing is no longer a bottleneck. There are fewer surprises when the code is delivered (see Figure 3).

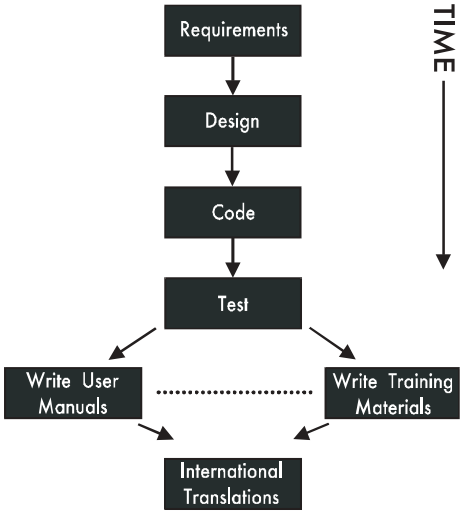


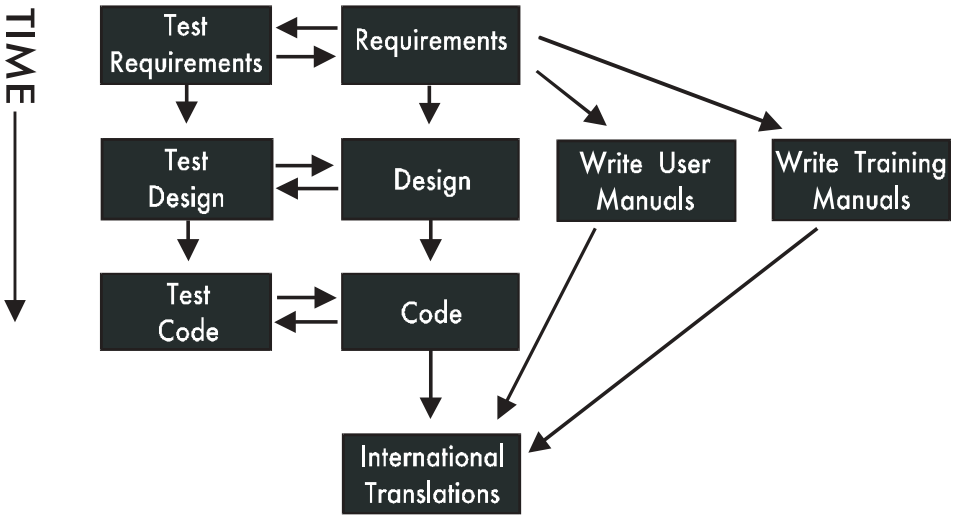
Figure 2: *Standard Development Life Cycle*

The RBT Methodology

The RBT methodology is a 12-step process. Each of these steps is described below.

1. **Validate requirements against objectives.** Compare the objectives, which describe *why* the project is being initiated, to the requirements, which describe *what is* to be delivered. The objectives define the success criteria for the project. If the *what* does not match the *why*, then the objectives cannot be met, and the project will not succeed. If any of the requirements do not achieve the objectives, then they do not belong in the project scope.
2. **Apply use cases against requirements.** Some organizations document their requirements with use cases. A use case is a task-oriented users' view of the system. The individual requirements, taken together, must be capable of satisfying any use-case scenarios; otherwise, the requirements are incomplete.
3. **Perform an initial ambiguity review.** An ambiguity review is a technique for identifying and eliminating ambiguous words, phrases, and constructs. It is not a review of the content of the requirements. The ambiguity review produces a higher-quality set of requirements for review by the rest of the project team.
4. **Perform domain expert reviews.** The domain experts review the requirements for correctness and completeness.
5. **Create cause-effect graph.** The requirements are translated into a cause-effect graph, which provides the following benefits:
   - It resolves any problems with aliases (i.e., using different terms for the same cause or effect).
   - It clarifies the precedence rules among the requirements (i.e., what causes are required to satisfy what effects).
   - It clarifies implicit information,

Figure 3: *Life Cycle With Testable Requirements and Integrated Testing*



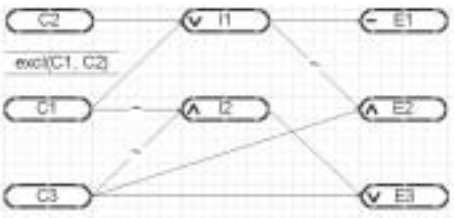


Figure 4: *Cause-Effect Graph*

making it explicit and understandable to all members of the project team.

- It begins the process of integration testing. The code modules eventually must integrate with each other. If the requirements that describe these modules cannot integrate, then the code modules cannot be expected to integrate. The cause-effect graph shows the integration of the causes and effects.

6. **Logical consistency checks performed and test cases designed.** A tool identifies any logic errors in the cause-effect graph. The output from the tool is a set of test cases that are 100 percent equivalent to the functionality in the requirements.
7. **Review of test cases by requirements authors.** The designed test cases are reviewed by the requirements authors. If there is a problem with a test case, the requirements associated with the test case can be corrected and the test cases redesigned.
8. **Validate test cases with the users/domain experts.** If there is a problem with the test case, the requirements associated with it can be corrected and the test case redesigned. The users/domain experts obtain a better understanding of what the deliverable system will be like. From a Capability Maturity Model® Integration℠ (CMMI℠) perspective, you are validating that you are *building the right system*.
9. **Review of test cases by developers.** The test cases are also reviewed by the developers. By doing so, the developers understand what they are going to be tested on, and obtain a better understanding of what they are to deliver so they can deliver for success.
10. **Use test cases in design review.** The test cases restate the requirements as a series of causes and effects. As a result, the test cases can be used to validate that the design is robust enough to satisfy the requirements. If the design cannot meet the requirements, then either the requirements are infeasible or the design needs rework.
11. **Use test cases in code review.** Each code module must deliver a portion of the requirements. The test cases can be used to validate that each code module delivers what is expected.
12. **Verify code against the test cases derived from requirements.** The final step is to build test cases from the logical test cases that have been designed by adding data and navigation to them, and executing them against the code to compare the actual behavior to the expected behavior. Once all of the test cases execute successfully against the code, then it can be said that 100 percent of the functionality has been verified and the code is ready to be delivered into production. From a CMMI perspective, you have verified that you are *building the system right*.

An Ambiguity Review

Here is a sample of a requirement written in first draft. It is not testable because it contains ambiguities.

> ATMs shall send an alert to the information technology (IT) department when the ATM has been tampered with. In the event that the ATM is opened without the key and security code, the ATM will alert the IT department immediately so the appropriate action can be taken.

After performing an ambiguity review of the requirements, the following ambiguities are identified:

- What type of alert does the ATM issue to the IT department?
- What is the definition of *tampered with*?
- Is tampered with the same as "in the event that the ATM is opened without the key and security code?"
- What happens if the key is used and an invalid security code is entered?
- What is the alert text?
- What is the *appropriate* action?

The requirements are revised so that the ambiguities are eliminated. The requirements are now testable.

> ATMs shall send a tamper alert to the IT department when the ATM has been tampered with, i.e., opened without the key and the valid security code.
>
> Case 1: (1) If the service operator enters the key into the ATM, then the following message displays on the ATM console: "Please enter the valid security code." (2) If the service operator enters the valid security code, then the ATM opens.
>
> Case 2: After entering the key in the ATM, if the service operator enters an incorrect security code, then (1) the following message displays on the ATM console: "Security Code invalid. Please re-enter." (2) The service operator now has three tries to enter the valid security code. If a valid security code is entered in less than or equal to three tries, then the ATM is opened. Each time an invalid security code is entered, the following message is displayed on the ATM console: "Security code invalid. Please re-enter."
>
> Case 3: If a valid security code has not been entered by the third try, then (1) the following message displays on the ATM console: "Security code invalid. The IT department will be notified." (2) The ATM alerts the IT department immediately.
>
> Case 4: In the event that the ATM is opened without the key and the valid security code, then the ATM sends a tamper alert to the IT department immediately.

A Cause-Effect Graphing Example

Consider a check-debit function whose inputs are *new balance* and *account type*, which is either postal or counter, and whose output is one of four possible values:

- Process debit and send out letter.
- Process debit only.
- Suspend account and send out letter.
- Send out letter only.

The function has the following requirements and is testable:

- If there are sufficient funds available in the account to be in credit, or the new balance would be within the authorized overdraft limit, then process the debit.
- If the new balance is below the authorized overdraft limit, then do not process the debit, and if the account type is postal, then suspend the account.
- If a) the transaction has an account type of postal or b) the account type is counter and there are insufficient funds available in the account to be in credit, then send out letter.

The causes for the function are as follows:

- C1 – New balance is in credit.
- C2 – New balance is in overdraft, but within the authorized overdraft limit.
- C3 – Account type is postal.

The effects for the function are as follows:

- E1 – Process the debit.
- E2 – Suspend the account.
- E3 – Send out letter.

A cause-effect graph shows the relationships between the conditions (causes) and the actions (effects) in a notation similar to that used by designers of hardware logic circuits. The check-debit requirements are modeled by the cause-effect graph shown in Figure 4. C1 and C2 cannot be true at the same time.

The cause-effect graph is converted into a decision table. Each column of the decision table is a rule. The table comprises two parts. In the top part, each rule is tabulated against the causes. A *T* indicates that the cause must be TRUE for the rule to apply and an *F* indicates that the condition must be FALSE for the rule to apply. In the bottom part, each rule is tabulated against the effects. A *T* indicates that the effect will be performed; an *F* indicates that the effect will not be performed; an asterisk (*) indicates that the combination of conditions is infeasible and so no effects are defined for the rule. The check-debit function has the decision table shown in Table 1.

Only test cases one through five in Table 1 are required to provide 100 percent functional coverage. Rule No. 6 does not provide any new functional coverage that has not already been provided by the other five rules, so a test case is not required for rule No. 6. No test cases are generated for rule Nos. 7 and 8 because they describe infeasible conditions since C1 and C2 cannot be true at the same time. The final set of test cases with sample-data values is described in Table 2.

Real-Life Problem Test Cases

With a real-life problem, there are usually far more than three inputs (causes). As an example, in one application where RBT was applied, there were 37 inputs. This allowed a maximum of 2**37, or 137,438,953,472 possible test cases. RBT resolved the problem with 22 test cases that provided 100 percent functional coverage.

Consider the following thought experiment: Put 137,438,953,450 red balls in a giant barrel. Add 22 green balls to the barrel and mix well. Turn out the lights. Pull out 22 balls. What is the probability that you have selected all 22 of the green balls? If this does not seem likely to you, try again. Return the balls and pull out 1,000 balls. What is the probability that you now have selected all 22 of the green balls? If this still does not seem likely to you, try again. Return the balls and pull out 1,000,000 balls. What is the probability that you now have selected all 22 of the green balls? This is what *gut-feel* testing really is.

For most complex problems it is impossible to manually derive the right combination of test cases that covers 100 percent of the functionality. The right combination of test cases is made up of individual test cases, and each covers at least one type of error that none of the other test cases covers. Taken together, the test cases cover 100 percent of the functionality. Any more test cases would be redundant because they would not catch an error that is already covered by an existing test case.

Gut-feel testing often focuses only on the normal processing flow. Another name for this is the *go path*. Gut-feel testing often creates too many (redundant) test cases for the go path. Gut-feel testing also often does not adequately cover all the combinations of error conditions and exceptions, i.e., the processing off the go path. As a result, gut-feel testing suffers when it comes to functional coverage.

Summary

In summary, the RBT methodology delivers maximum coverage with the minimum number of test cases. This translates into 100 percent functional coverage and approximately 70 percent to 90 percent code coverage. RBT also provides quantitative test progress metrics within the 12 steps of the RBT methodology, ensuring that testing is adequately provided and is no longer a bottleneck. Logical test cases are designed and become the basis for highly portable capture/playback test scripts.◆

Table 2: *Test Cases*

	CAUSES						EFFECTS
Test Case	*Current Balance*	*Debit Amount*	*Difference*	*Overdraft Limit*	*New Balance*	*Account Type*	*Action*
1	-$70	$50	-$120	-$100	-$70	Counter	**Send out letter.**
2	$420	$2,000	-$1,580	-$1,500	$420	Postal	**Suspend the account; send out letter.**
3	$650	$800	-$150	-$250	-$150	Counter	**Process the debit; send out letter.**
4	$2,100	$1,200	$900	-$1,000	$900	Counter	**Process the debit.**
5	$250	$150	$100	-$500	$100	Postal	**Process the debit; send out letter.**

Rules	1	2	3	4	5	6	7	8
C1: New balance is in credit.	F	F	F	T	T	F	T	T
C2: New balance is in overdraft, but within the authorized limit.	F	F	T	F	F	T	T	T
C3: Account is postal.	F	T	F	F	T	T	F	T
E1: Process the debit.	F	F	T	T	T	T	*	*
E2: Suspend the account.	F	T	F	F	F	F	*	*
E3: Send out letter.	T	T	T	F	T	T	*	*

Table 1: *Decision Table*

Note

1. A functional requirement specifies what the system must be able to do in terms that are meaningful to its users. A nonfunctional requirement specifies an aspect of the system other than its capacity to do things. Examples of nonfunctional requirements include those relating to performance, reliability, serviceability, availability, usability, portability, maintainability, and extendibility.

Did this article pique your interest?

You can hear more from Gary E. Mogyorodi at the Fifteenth Annual Software Technology Conference Apr. 28-May 1, 2003 in Salt Lake City, UT. He will be presenting in Track 1 on Monday, Apr. 28, at 3:10 p.m.

About the Author

Gary E. Mogyorodi is a senior consultant with Bloodworth Integrated Technology, Inc., consulting, training, and mentoring in software testing, and specializing in requirements-based testing. He has more than 29 years of experience in the computing industry and has presented at numerous conferences, including the Software Technology Conference, Software Quality Forum, Toronto SPIN, Starwest, and more. Mogyorodi has a bachelor's degree in mathematics from the University of Waterloo, and a master's degree in business administration from McMaster University.

Bloodworth Integrated Technology, Inc.
36A Mendota Road #8
Toronto, Ontario
Canada M8Y 1E8
Phone: (416) 521-7200
Fax: (419) 831-6407
E-mail: garym@bitspi.com

Determining Return on Investment Using Software Inspections

Don O'Neill
Consultant

This article examines the defined measurements used to form the derived metric for return on investment. These measurements involve additional cost multiplier, defect detection rate, cost to repair, and detection cost. This article further examines the behavior of these measurements and metrics for various software product-engineering styles using data collected in the National Software Quality Experiment.

To deliver superior quality, many organizations have made commitments to initiatives on the Software Engineering Institute's (SEI) Capability Maturity Model® (CMM®), ISO 9001, or Six Sigma. Each of these initiatives has one thing in common: software inspections.

Managers are interested in knowing the return on investment to be derived from software process improvement actions. The software inspection process gathers some of the data needed to determine this [1]. Software inspections are structured to serve the needs of quality management in verifying that a software artifact complies with its standard of excellence for software engineering. The focus is on verification, on doing the job right. The software inspection is a formal review held at the conclusion of a life-cycle activity and serves as a quality gate with exit criteria for moving on to subsequent activities [2].

The National Software Quality Experiment (NSQE) is a mechanism for obtaining core samples of software product quality. The NSQE includes a micro-level national database of product quality populated by a continuous stream of samples from industry, government, and military services. The centerpiece of the experiment is the software inspection lab where data collection procedures, product checklists, and participant behaviors are packaged for operational project use. The NSQE is providing valuable insights on the practice of software inspections through its database of thousands of software inspection sessions from dozens of organizations containing tens of thousands of defects along with the pertinent information needed to pinpoint their root causes [3].

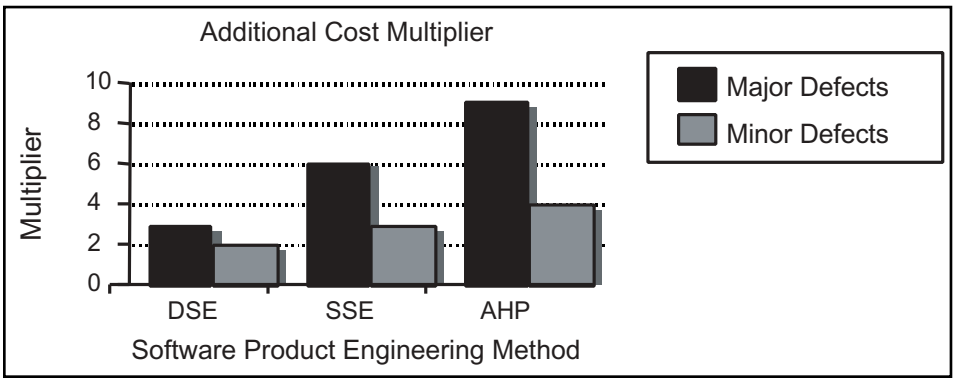
> *"The NSQE is providing valuable insights on the practice of software inspections through its database of thousands of software inspection sessions from dozens of organizations containing tens of thousands of defects along with the pertinent information needed to pinpoint their root causes."*

To review NSQE description and data summaries, please visit the Web resource listed in [4].

The model in this article for return on investment bases the savings on the cost avoidance associated with detecting and correcting defects earlier rather than later in the product evolution cycle. It is defined as *net savings* divided by *detection cost*, where net savings is *cost avoidance* less *cost to repair now*; detection cost is the cost of preparation effort and the cost of conduct effort. Savings result from early detection and correction, which avoids the increased cost multiplier associated with detection and correction of defects later in the life cycle.

A major defect that leaks from development to test may cost two to 10 times more to detect and correct than if done earlier. Some of these defects leak further from test to customer use and may cost an additional two to 10 times to detect and correct. A minor defect may cost an additional two to four times to correct later. The defined measurements collected in the software inspection lab may be combined in complex ways to form the derived metric for return on investment. These involve an additional cost multiplier, defect detection rate, cost to repair, and detection cost.

Figure 1: *Additional Cost Multiplier*



Software Product Engineering Method

The values for these complex parameters revolve around the Software Product Engineering (SPE) key process area being practiced, which is a CMM Level 3 key process area. Three levels of achievement of SPE are identified as follows:

1. Ad-hoc programming is characterized by a code-and-upload life cycle and a hacker coding style. This is common in low software process maturity organizations, especially those facing time-to-market demands.
2. Structured software engineering employs structured programming, modular design, and defined programming style, and pays close attention to establishing and maintaining traceability among requirements,

specification, architecture, design, code, and test artifacts. This is the minimum expectation for CMM Level 3 [5].
3. Disciplined software engineering is more formal and might be patterned after cleanroom software engineering, Personal Software Process, Team Software Process, and extreme programming techniques [6, 7, 8]. This is the expectation for SEI CMM Level 4 and 5 organizations [5].

Additional Cost Multiplier

Since savings result from avoiding the increased cost multiplier associated with detection and correction of defects later in the life cycle, the question of the cost multiplier must be answered to determine the return on investment.

Some set the additional cost multiplier for finding and fixing a defect detected after delivery at 100 times earlier detection [9]. Others have measured it more precisely and found it to be 10 times more for each life-cycle activity. IBM Rochester, Rochester, Minn., winner of the Malcolm Baldrige National Quality Award in 1990, reported that defects leaking from code to test cost nine times more to detect and correct, and defects leaking from test to the field cost 13 times more to detect and correct [10].

Why Is There a Multiplier?

An example may help illustrate why a leaked defect costs more. A code defect that leaks into testing may require multiple test executions to confirm the error and additional executions to obtain debug information. Once a leaked defect has been detected, the producing programmer must put aside the task at hand, refocus attention on correcting the defect and confirming the correction, and then return to the task at hand. The corrected artifact must then be reinserted into the SPE or product release pipeline and possibly into user operations.

Keep in mind that software changes experience high defect rates. In addition, the span of impact from defects introduced during different activities of the life cycle varies. The number of source lines affected by a requirement defect might exceed 100 lines; by a design defect, 10 to 100 lines; by a coding defect, less than 10 lines; and by a clerical defect, one line.

What Is the Multiplier?

It is reasonable to expect the additional cost multiplier to be linked to the SPE method practiced. Figure 1 portrays the additional cost multiplier by SPE method.

1. Ad-hoc programming (AHP) is likely to experience a multiplier of eight to 10 times in detecting and correcting major defects in spaghetti-bowl coding that lacks order and consistency. The multiplier for minor defects is likely to be four times.
2. Structured software engineering (SSE) is likely to experience a multiplier of five to seven times in detecting and correcting major defects in the production of well structured, consistently recorded components with organized relationships among modules and traceability among life-cycle artifacts. The multiplier for minor defects is likely to be three times.
3. Disciplined software engineering (DSE) with its formal focus on quality may experience a multiplier of two to four times in detecting and correcting major defects. The multiplier for minor defects is likely to be two times.

What Effect Does the Multiplier Have?

In summary, an undetected major defect that leaks to the next phase of the life cycle may cost two to 10 times more to

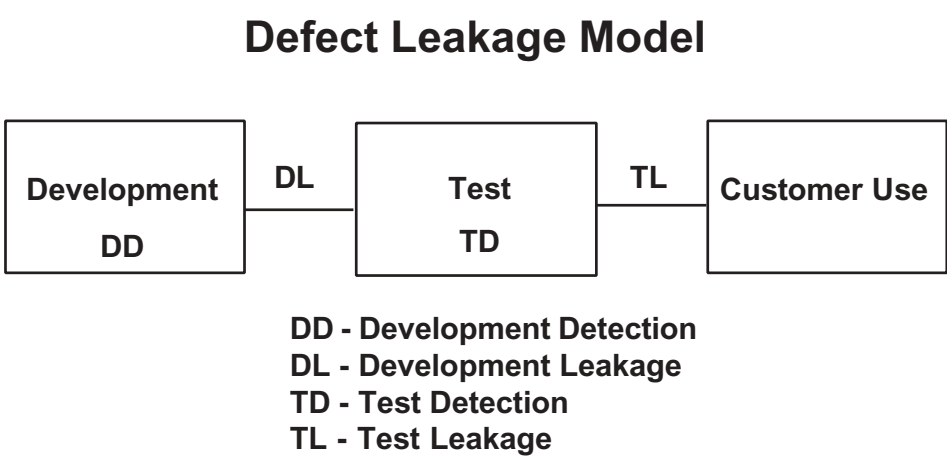


Figure 2: *Defect Leakage Model*

Figure 3: *Development Detection*

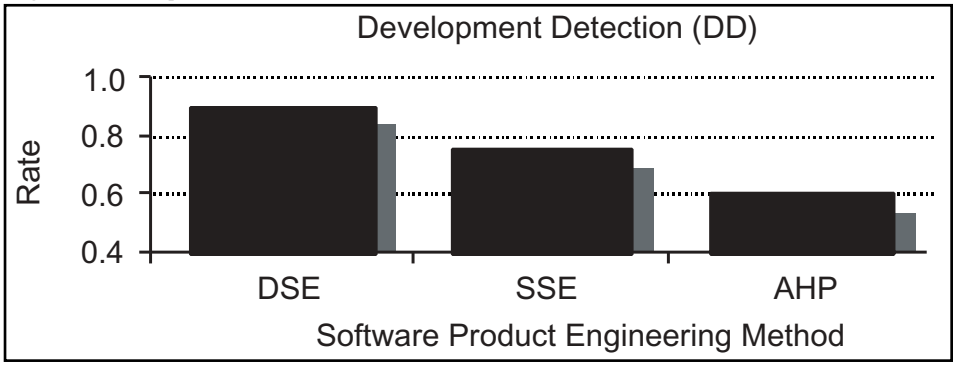
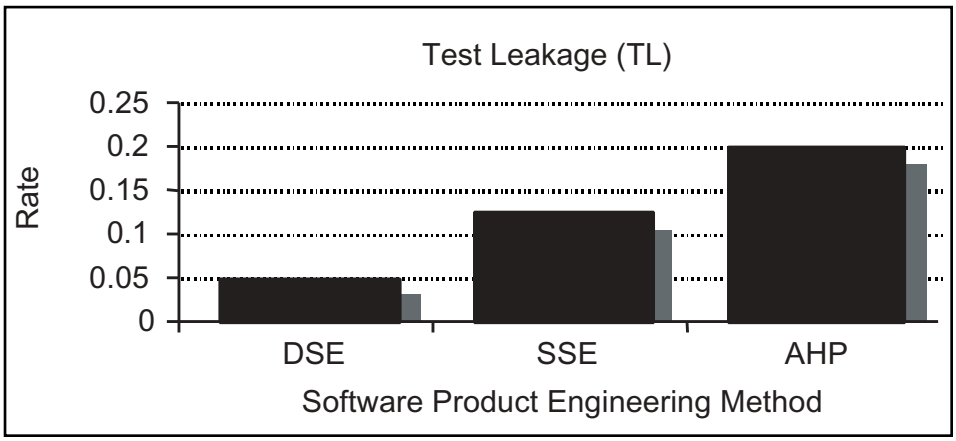


Figure 4: *Test Leakage*



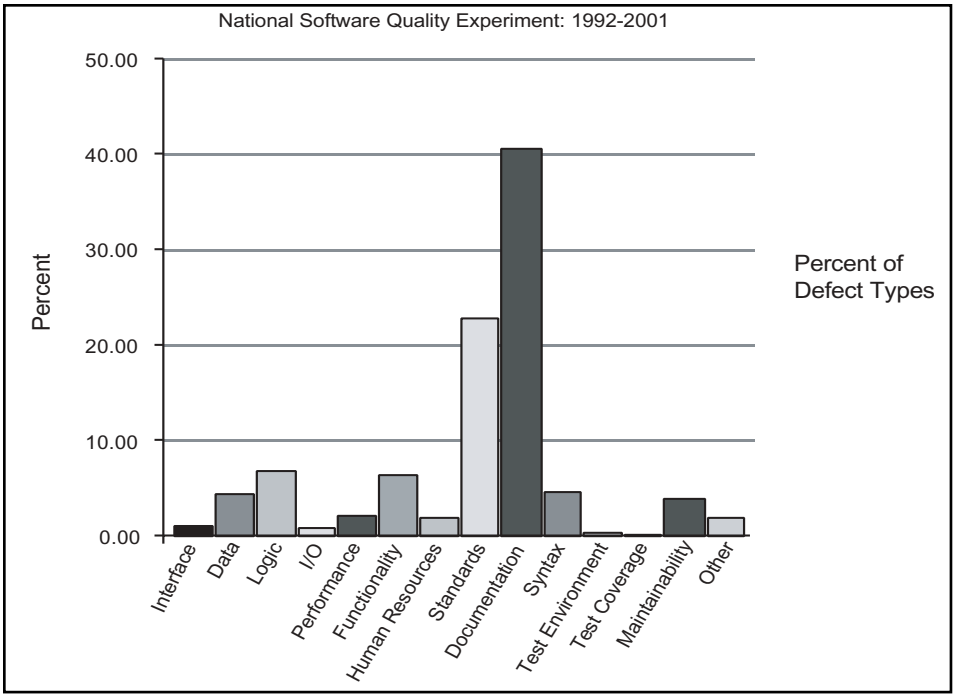


Figure 5: *Defect Type Distribution*

detect and correct. A minor defect may cost two to four times more to detect and correct. The resulting net savings then may be up to nine times for major defects and up to three times for minor defects.

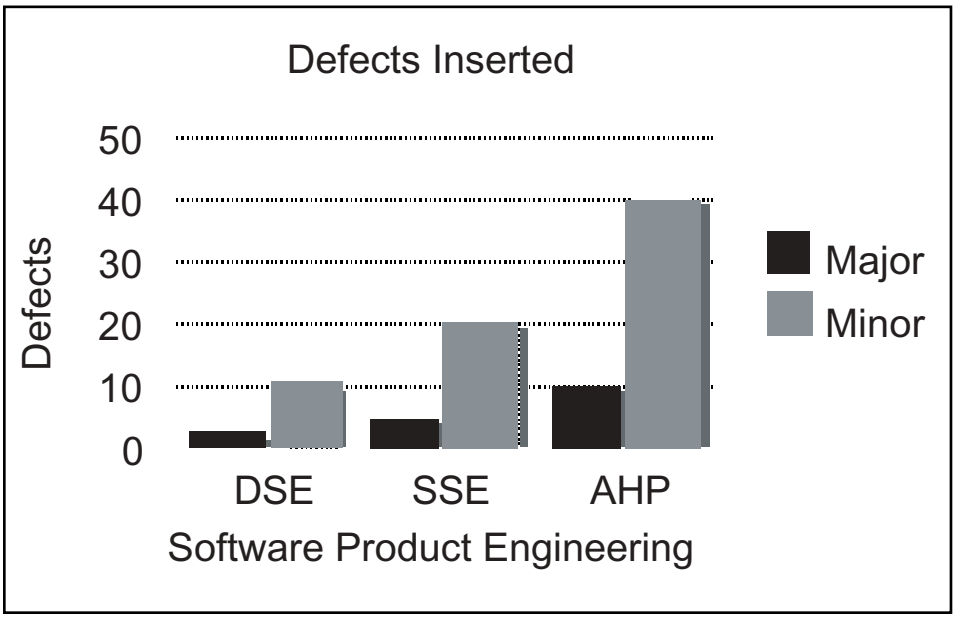
Defect Detection Rate

The model shown in Figure 2 illustrates that defects are detected in development (DD) and test (TD), defects leak from development (DL), and defects leak from test (TL). Defect detection rate equals the number of defects detected divided by the number of defects present.

It is reasonable to expect the defect detection rate to be linked to the SPE method practiced, including the software inspection process followed. Figures 3 and 4 illustrate DD and TL using empirically derived values for the defect leakage model factors of each SPE method. While the defect DD rates are based on the results of the NSQE, the expected TD uses a notional value in order to complete the analysis.

1. AHP is likely to experience a defect DD rate in the range of 0.50 to 0.65. While the TL depends on the adequacy of the test process, AHP is likely to experience TL in the range of 0.175 to 0.25 based on an expected TD of 0.50.
2. The SSE is likely to experience a defect DD rate in the range of 0.70 to 0.80, and a TL in the range of 0.1 to 0.15 based on an expected TD of 0.50.
3. DSE may experience a defect DD rate in the range of 0.85 to 0.95, and a TL in the range of 0.025 to 0.075 based on an expected TD of 0.50 [4].

Figure 6: *Defects Inserted Per Thousand Lines*



Cost to Repair

The cost to repair a defect detected in the life-cycle activity in which it was inserted depends on the SPE method practiced and the business environment in which it is operating. This must be supplied by the organization based on its actual cost history and the superior knowledge of its personnel.

In determining the cost to repair, the organization needs to obtain this cost-by-defect type. During the software inspection lab session, each detected defect is assigned a type, including interface, data, logic, input/output, performance, functionality, human factors, standards, documentation, syntax, maintainability, and others. The defect type distribution revealed by the NSQE is shown in Figure 5 [3, 11, 12].

For purposes of the software inspections' return-on-investment analysis, the cost-to-repair factor is included in the expression for net savings discussed later. For analysis here, the cost to repair is set at one hour for a major defect and one hour for a minor defect.

Major defects are those that affect execution; minor defects do not affect execution but may still be important. Some practitioners mistakenly assign a major classification to defects that require extensive rework, and a minor classification to defects that require little rework. When this mistake is made, the major rework metric will systematically exceed the minor rework metric.

In actual practice, as measured by the NSQE during the past 10 years, certain major defects such as eliminating magic numbers or inserting a *when others clause* require one or two line changes in the code. Also, some minor defects such as lack of traceability or adding prologue and version history commentary may have more pervasive impact.

During software inspections, many defects are detected; many are trivial and require little cost-to-repair effort. Others may be more complex and require substantial effort. Where the organization has superior knowledge of the cost-to-repair metric, it should use that informa-

tion. Where the organization lacks measured results, using one hour for cost to repair is an initial value that many have found valid.

Defect Detection Cost

The cost of defect detection includes the participants' efforts to prepare and conduct the software inspection. Time to conduct the inspection includes the actual physical time consumed by the software inspection meeting. Effort to conduct the inspection includes the actual time it takes to complete the inspection multiplied by the number of participants. Factors used to determine detection cost include the size of the artifact being inspected, the number of defects inserted, and the relationship between the effort to prepare and conduct the inspection.

It is reasonable to expect the defect detection cost to be linked to the SPE method practiced, including the software inspection process followed. Figure 6 illustrates the following defect insertion rates by the SPE method.

1. AHP may experience a preparation effort divided by a conduct effort ratio of approximately 0.60 in inspecting artifacts of 400 to 600 lines of code, as experienced by CMM Level 1 organizations in the NSQE [6, 8, 10]. These organizations may experience a defect insertion rate of 40 to 60 defects per thousand lines of code.
2. SSE may experience a preparation effort divided by a conduct effort ratio of approximately 0.80 in inspecting artifacts of 200 to 400 lines of code, as experienced by CMM Level 3 organizations in the NSQE [6, 8, 10]. These organizations may experience a defect insertion rate of 20 to 30 defects per thousand lines of code.
3. DSE may experience a preparation effort divided by a conduct effort ratio of approximately 1.0 in inspecting artifacts of less than 200 lines of code. These organizations may experience a defect insertion rate of 10 to 15 defects per thousand lines of code.

Reasoning About ROI

Software inspections' return on investment is equal to net savings divided by detection cost. Evaluating the following expression assists in reasoning about return on investment:

ROI = Net Savings/Detection Cost

Reasoning About Net Savings

Net savings is equal to cost avoidance minus cost to repair now. Evaluating the following expression assists in reasoning about net savings:

Net Savings =
Cost Avoidance - Cost to Repair Now

Cost avoidance results from avoiding the higher costs that occur from deferred detection and correction. The additional cost multiplier comes into play in the following ways:

- M1 is the additional cost-to-repair multiplier for development to test major defect leakage.
- M2 is the additional cost-to-repair multiplier for test to customer-use major defect leakage.
- M3 is the additional cost-to-repair multiplier for minor defect leakage.

Evaluating the following expression assists in reasoning about cost avoidance:

Cost Avoidance =
Major Defects x {(M1 x DD) + (M1 x DD) x (M2 x TL) x C1} + Minor Defects x M3

The cost to repair now, simply the cost of defect correction, is subtracted from cost avoidance to yield net savings. Evaluating the following expression assists in reasoning about net savings:

Net Savings = Major Defects x {(M1 x DD) + (M1 x DD) x (M2 x TL) x C1 - C1} + Minor Defects x (M3 - C2)

Simplifying the expression results in the following:

Net Savings =
Major Defects x {C1 x [(M1 x DD) x (1 + (M2 x TL))] - 1} + Minor Defects x (M3 - C2)

Where:

- M1: (2 - 10) Additional Cost-to-Repair Multiplier for Development to Test Major Defect Leakage.
- M2: (2 - 10) Additional Cost-to-Repair Multiplier for Test to Customer Use Major Defect Leakage.
- M3: (2 - 4) Additional Cost to Repair for Minor Defect Leakage.
- DD: (0.5 - 0.95) Defect Detection Rate for Development to Test.
- TL: (0.05 - 0.5) TL Rate for Test to Customer Use.
- C1: Average Cost to Repair Major Defect (in hours of effort).
- C2: Average Cost to Repair Minor Defect (in hours of effort).

Reasoning About Detection Cost

Detection cost is equal to preparation effort plus conduct effort. Evaluating the following expression assists in reasoning about detection cost:

Detection Cost =
Preparation Effort + Conduct Effort

Preparation effort is the total minutes of preparation effort. Conduct effort is the minutes of conduct time multiplied by the number of participants. Substituting the resulting expression is the following:

Detection Cost =
{Minutes of Preparation Effort + (Minutes of Conduct Time x Participants)}/60

Where:

- Participants: (4-6) Number of participants.
- 60 minutes per hour.

A Worked Example

The return on investment is determined by using the expression for net savings specified above and setting the parameters for cost-to-repair multiplier, defect detection, and defect leakage. For example, to determine the expression for return on investment to be used in a project spreadsheet, the following example is offered:

1. Setting the parameters: M1=5, M2=10, M3=2, DD=0.6, TL=0.25, C1=1, and C2=1.
2. Using the expression:

   Net Savings = Major Defects x {(M1 x DD) + (M1 x DD) x (M2 x TL) x C1 - C1} + Minor Defects x (M3 - 1)

3. Substituting for the values of the worked example:

   Net Savings = Major Defects x {(5 x .6) + (5 x .6) x (10 x .25) x 1- 1} + Minor Defects x (2-1)

4. The following expression for Net Savings results:

   Net Savings = 9.5 x Major Defects + Minor Defects

The result of the worked example is

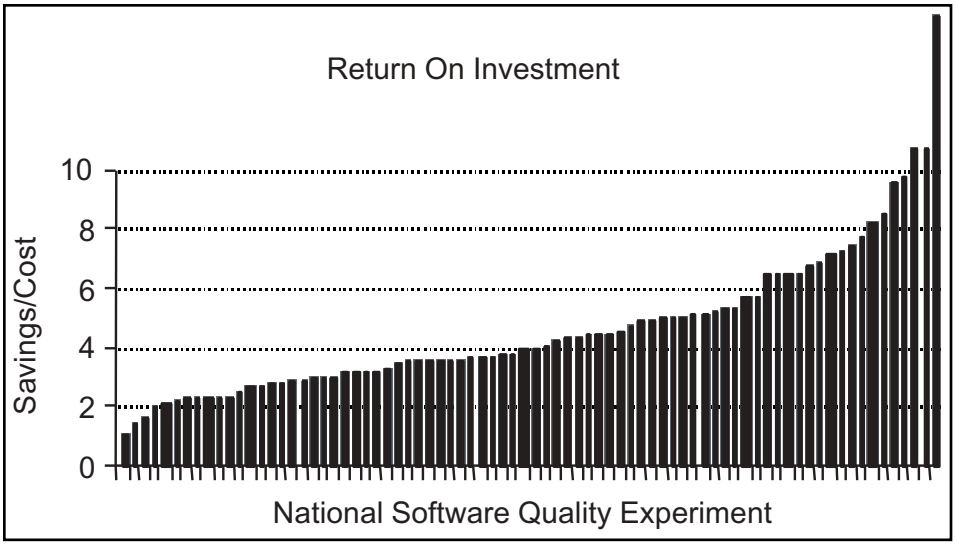


Figure 7: *Return on Investment*

a simplified expression for net savings of the type used to derive the return-on-investment metric in the NSQE. Figure 7 illustrates the range of practice for return on investment.

Selecting Parameter Values

Where an organization possesses superior knowledge of its software operation, it should utilize the parameter values that best reflect this understanding. Candidate parameter values for each SPE method are shown in Table 1 for DSE, SSE, and AHP.

Computing Return on Investment

Software process improvement goals involve both cost and quality. The achievement of these goals varies according to the SPE method practiced, and these variations are illustrated in the application of the selected parameter values (see Table 2). AHP practitioners derive substantial net savings and return on investment, but a high incidence of defect leakage into customer use. The SSE practitioners experience attractive net savings and return on investment, and a reduced defect leakage into customer use. DSE practitioners barely recoup the investment but achieve a very low incidence of defect leakage into customer use.

Transition From Cost to Quality

In using software inspections, the goals vary with the SPE method used, transitioning from cost to quality.

By necessity, the focus of AHP practitioners is on reducing cost by detecting as many defects as possible. With 40 to 60 defects inserted per thousand lines of code, a defect detection rate of 0.5 to 0.65, and an additional cost multiplier of eight to 10, the result is a net savings of 234.8 to 285 labor hours and a defect leakage expectation of 8.75 to 12.5 per thousand lines of code, numbers that promote a focus on cost. For this group, finding defects is like finding free money, and there are always more defects to find; however, managers struggle to meet cost and schedule commitments.

The SSE focus is split between reducing cost and improving quality. With 20 to 30 defects inserted per thousand lines of code, a defect detection rate of 0.70 to 0.80, and an additional cost multiplier of five to seven, the result is a net savings of 65 to 85.23 labor hours and a defect leakage expectation of 2.5 to 3.75 per thousand lines of code, numbers that promote an attraction to both goals. For this group, there is constant dithering between cost and quality.

Without question, the focus of DSE practitioners is on eliminating every possible defect even if defect detection costs exceed net savings and the return on investment falls below the break even point. With 10 to 15 defects inserted per thousand lines of code, a defect detection rate of 0.85 to 0.95, and an additional cost multiplier of two to four, the result is a net savings of 12.49 to 18.55 labor hours and a defect leakage expectation of 0.3125 to 0.9375 per thousand lines of code, numbers that promote a focus on quality. For this group, every practitioner is riveted on achieving perfection.

Table 1: *Candidate Parameter Values*

	M1	M2	M3	Major Per K	Minor Per K	DD	TL	Prep Min	Conduct Min	Participants
DSE	2-4	2-4	2	2.5	10	0.95-0.85	0.025-0.0075	500	120	4
SSE	5-7	5-7	3	5	20	0.70-0.80	0.075-0.150	400	120	4
AHP	8-10	8-10	4	10	40	0.50-0.65	0.175-0.250	300	120	4

Table 2: *Computing Return on Investment*

	DD	M1	TL	M2	M3	Net Savings	Detection Cost	ROI Per K	Leaks
DSE Disciplined Software Engineering	0.95	2	0.025	2	2	12.49	16.33	0.76	0.3125
	0.90	3	0.050	3	2	15.26	16.33	0.93	0.6250
	0.85	4	0.075	4	2	18.55	16.33	1.14	0.9375
SSE Structured Software Engineering	0.80	5	0.100	5	3	65.00	14.67	4.43	2.5000
	0.75	6	0.125	6	3	74.38	14.67	5.07	3.1250
	0.70	7	0.150	7	3	85.23	14.67	5.81	3.7500
AHP Ad Hoc Programming	0.65	8	0.175	8	4	234.80	13.00	18.06	8.7500
	0.60	9	0.200	9	4	261.20	13.00	20.09	10.0000
	0.55	10	0.225	10	4	288.75	13.00	22.21	11.2500
	0.50	10	0.250	10	4	285.00	13.00	21.92	12.5000

Summary

In studying the issues associated with the new realities of the workplace and the new software engineering responses to the issues, the answer lies in first understanding what the enterprise is trying to do, and then in how the enterprise does it.

What the enterprise is trying to do revolves around the management of commitments and the drive toward product perfection. The management of commitments is primarily associated with time to market but also performance to budget. The drive toward product perfection is associated with satisfying and delighting the customer with a continuous stream of the right capabil-

ities and features packaged in a defect-free container.

How the enterprise does this revolves around its software product engineering practice. The three modes of practice in software product engineering include AHP, SSE, and DSE.

What is actually occurring is a competition among stresses, not all of which can be satisfied. In reasoning about new software engineering, it is important to explicitly acknowledge that choices must be made. The drive toward perfection clashes with the drive to achieve time to market. In the short-term environment of today, the successful enterprise makes the strategic selection and accepts any collateral damage.

When an organization has superior knowledge of the parameter values for software inspections return on investment, it is able to derive its own return-on-investment metric. To perform this computation, simply visit the tool at <http://members.aol.com/ONeillDon/nsqe-roi.html>.◆

References

1. McGibbon, T. "A Business Case for Software Process Improvement." Rome Laboratory DACS Report, 30 Sept. 1996.
2. O'Neill, Don. "Peer Reviews." Encyclopedia of Software Engineering. Wiley Publishing, Inc., Jan. 2002.
3. O'Neill, Don. "National Software Quality Experiment: A Lesson in Measurement 1992-1997." CrossTalk 11.12 (Dec. 1998) <www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1998/12/oneill.asp>.
4. O'Neill, Don. National Software Quality Experiment Description and Data Summaries <http://members.aol.com/ONeillDon/nsqe-results.html>.
5. Paulk, Mark C. The Capability Maturity Model: Guidelines for Improving the Software Process. Reading, MA: Addison-Wesley, 1995: 270-276.
6. Prowell, Stacy J., Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. Cleanroom Software Engineering: Technology and Process. Addison Wesley Longman, 1999: 17, 33-90.
7. Humphrey, Watts. Introduction to the Personal Software Process. Reading, MA: Addison-Wesley, 1997.
8. Wells, J. Donovan <www.extremeprogramming.org>.
9. Basili, Vic, and Barry Boehm. "Top Ten Defect Reduction List." IEEE Software Jan. 2001.
10. Lindner, Richard J., and D. Tudahl. Software Development at a Baldrige Winner. Proc. of ELECTRO '94, Boston, MA, 12 May 1994: 167-180.
11. O'Neill, Don. "National Software Quality Experiment: Results 1992-1999." Software Technology Conference, Salt Lake City, UT, 1995, 1996, and 2000.
12. O'Neill, Don. National Software Quality Experiment: A Lesson in Measurement 1992-1997. First International Software Assurance Certification Conference. Chantilly, VA, 1 Mar. 1999: 1-14.

About the Author

Don O'Neill is a 43-year veteran software engineering manager and technologist currently serving as an independent consultant. He conducts defined programs for managing strategic software improvement, including implementing an organizational software inspections process, directing the National Software Quality Experiment, implementing software risk management on the project, conducting the Project Suite Key Process Area Defined Program, and conducting Global Software Competitiveness Assessments. O'Neill is a founding member of the Washington, D.C.-based Software Process Improvement Network and the National Software Council and serves as the executive vice president of the Center for National Software Studies. He has a bachelor's of science degree in mathematics form Dickinson College, and has completed a three-year residency at Carnegie Mellon University's Software Engineering Institute.

9305 Kobe Way
Montgomery Village, MD 20886
Phone: (301) 990-0377
E-mail: oneilldon@aol.com

A Pair Programming Experience

Dr. Randall W. Jensen
Software Technology Support Center

Agile software development methods, including extreme programming, have risen to the forefront of software management and development interest during the last few years. The "Agile Manifesto" published in 2001 created a new wave of interest in the agile philosophy and re-emphasized the importance of people, along with the idea of "pair programming." As defined, pair programming is two programmers working together, side by side, at one computer collaborating on the same analysis, design, implementation, and test. I was introduced to teamwork and pair programming indirectly as an undergraduate electrical engineering student in the 1950s. Later in 1975, I was asked to improve programmer productivity in a large software organization. The undergraduate experience led me to an experiment in pair programming. The very positive results of this experiment are the subject of the case study in this article.

Agile methods and extreme programming have risen to the forefront of software management and development interest during the last few years. Two definitions of *agile* are (1) able to move quickly and easily, and (2) mentally alert. Both definitions rely on the capabilities of the people within the development process.

The "Agile Manifesto" [1] published in *Software Development* in 2001 created a new wave of interest in the agile philosophy and reemphasized the importance of people. One of the points highlighted in the manifesto is, "We value individuals and interactions over processes and tools." That does not mean processes and tools are evil. It implies that individuals and interactions (people) are of higher priority than processes and tools.

Textbooks [2, 3] describe the importance of people in these new software development approaches that have demonstrated improved productivity and product quality. *Extreme programming* (XP) [4] is one member covered by the umbrella of agile methods. *Pair programming* [5] is a major practice [6] of XP. The official definition of pair programming is two programmers working together, side by side, at one computer collaborating on the same analysis, design, implementation, and test. In other words, consider it like two programmers using one pencil.

We have all experienced elements of the pair-programming concept in one way or another during our lives. How many times have you been stuck removing an error from a design or program with no success? When everything else failed, you went to your neighbor programmer, the *casual observer*, to see if you could get some assistance. While explaining the problem, you have a flash of inspiration, and the problem is quickly solved. How much time did you waste before asking a neighbor for insight? Can you relate this to pair programming?

I was introduced to pair programming indirectly as an undergraduate electrical engineering student in the 1950s. The class and laboratory workload were such that any free time during the four-year program was more wishful thinking than reality. Working part time made the program even more daunting. Fortunately, two other electrical engineering students in the same academic program were struggling with different sets of outside commitments. We decided to work together on homework assignments, lab work, and test preparation to lighten the course load.

> *"The second major benefit demonstrated in this experiment – a three order-of-magnitude improvement in error rate – is hard to ignore."*

We successfully maintained this approach through the entire program in spite of having been conditioned throughout our lives to perform solitary work. Our educational system does not condone or encourage teamwork. That education philosophy supports individual student evaluation, but works against learning. The teamwork concept became ingrained in my thinking as well as in my programming and management research activities.

Much later, I was asked to find ways to improve programmer productivity in a large software organization. The undergraduate experience led me to propose an experiment in the application of what we called *two-person programming teams*. The term pair programming had not been coined at that time. The experiment results are the subject of the remainder of this article.

Development Task

Problem

Providing a description of the results achieved through pair programming without knowledge of the project or development task underlying the experience would be meaningless. The software to be developed in this project was a multitasking real-time system executive. The product consisted of six independent components containing a total of approximately 50,000 source lines of code. The product contained no reused or commercial-off-the-shelf components. Fortran was the required software development language. The real-time executive was to be used to support the development of a large, complex software system by the developing organization. The development schedule for the executive was critical and short.

Team Composition

The development team consisted of 10 programmers with a wide range of experience and one manager. I tend to divide managers into two primary groups: Theory X[1] and Theory Y[2] [7, 8]. The manager for this task was experienced and from the Theory Y group.

The 10 programmers assigned to the executive development had prior experience that ran the gamut from an expert system programmer to a couple of fresh, young college graduates. None of these programmers had any experience working in a team environment. As a collection, I would place them as about average for that development organization.

The manager grouped the programmers into five teams according to their experience level. Each team pair was composed of the most experienced and least experienced programmer of the remaining

group. The first team consisted of the expert system programmer and a person who had just returned from a six-year leave of absence. The fifth team consisted of two programmers of near equal capability and experience. These first and fifth programming teams were important in the way they impacted the project. I will address their impacts in the Lessons Learned section of this article.

No special changes from normal were made to the development environment. The facilities were essentially two-person cubicles. The programming pairs were collocated in these cubicles. Each cubicle contained two computer workstations, two desks, and a common worktable. The pair-programming approach dictated that the pair (remember: two programmers, one pencil) use only one development terminal located on the common worktable. The second terminal was to be used for documentation, etc., not related to the team's assigned development.

One programmer of the pair functioned as the *driver* operating the keyboard and mouse, while the second programmer functioned more as a *navigator* or *co-pilot*. The navigator reviewed, in real time, the information entered by the driver. The roles of the two programmers were not permanent; frequent role changes occurred daily. The navigator was not a passive role at any time.

Results

A Priori

Project individuals could not directly obtain a productivity and error baseline for the project, but data was available from past projects that allowed them to project productivity and error averages for the project. The average productivity and error rates in most organizations with consistent management style and processes are near constant and quite predictable. The baseline productivity was determined to be approximately 77 source lines per person-month. The error rate for the development organization was normal for the aerospace industry. The numerical error rate value is not significant for this presentation, and will remain unknown.

Formal design walkthroughs and software inspections were not scheduled for this project. It would follow a classic waterfall development approach, which is inconsistent with today's agile methods. Formal preliminary and critical design reviews, as well as a final qualification test were planned. Formal review and test documentation were reduced to essential information; that is, all elements necessary to proceed with the development.

Topic	Historical	Pair Results	Gain
Productivity (lines/person-month)	77	175	127 percent
Error Rate			0.001 × normal

Table 1: *Pair Programming Productivity and Error Rate Gains*

A Posteriori

The productivity achieved in the real-time executive development was 175 source lines per person-month as shown in Table 1. We hoped for a productivity gain of anything greater than 0 percent. Any small gain would have compensated for the two programmers loading on each task. The 127 percent gain achieved was phenomenal and a cause for celebration.

The error analysis showed the project had achieved an error rate that was three orders of magnitude less than normal for the organization. Integration of the first two components (approximately 10,000 source lines) was completed with only two coding errors and one design error. The third component was integrated with no errors. The remaining three components had more errors, but the number of errors for these components was significantly less than normal.

The *continuous walkthrough* assumption was demonstrated to be very effective and more than compensated for the lack of formal walkthroughs. The formal preliminary and critical design reviews, as well as a final qualification test, were effective in keeping the five teams coordinated. Few problems were uncovered in the review and test activities.

After the experiment was completed, the development manager presented the very positive results to the organization's management staff. The project managers' reaction to the results was memorable – they claimed that their senior programmers would quit before they would team with another programmer. The use of pair programmers was never implemented in that organization.

Lessons Learned

Several positive and some negative characteristics were observed during the pair-programming experiment. In general, the attributes of the college experience were exhibited here. The positive attributes, not necessarily in any order, are as follows:

- **Brainstorming.** According to the programmers, active real-time collaboration produced higher quality designs than would have been achieved working alone. Little time was lost optimizing code with more than one brain working.
- **Continuous Design Walkthrough.** The design and code were reviewed in real time by both programmers who ultimately produced fewer errors in each team product. Classic walkthroughs and inspections are, whether we like it or not, somewhat adversarial. The continuous walkthroughs within the team were more positive and supportive.
- **Focused Energy.** The individual teams appeared to be more focused in their activities. The highly visible aspect of this attribute was that programmers took fewer breaks for restrooms, coffee, outside discussions, etc.
- **Mentor.** When we started work in this industry, we were usually told about on-the-job training that never materialized. Pair programming, when the two programmers were not of the same experience level, provided a craftsman/apprentice relationship that elevated the junior programmer's skill quickly. Conversely, the craftsman's skill is extended by the apprentice's questions and thinking outside of the box.
- **Motivation.** In general, the programming pairs appeared much more motivated than their single counterparts. The motivation level cannot be solely attributed to the pair concept or the experiment itself. Some of the motivation must be attributed to the project manager. Some must be attributed to rapid progress and the product quality. One of the Theory Y assumptions is that motivation occurs at the social, esteem, and self-actualization levels, as well as physiological and security levels.
- **Problem Isolation.** The time wasted with two pairs of eyes (or brains) was significantly less than the amount of time wasted trying to solve a problem in isolation.

Conversely, the negative observations cannot be ignored. The important observations, not necessarily in order of importance, are as follows:

- **Counter-Productivity.** Pairing programmers of the same experience and capability level is often counter-productive. The most troublesome pairs we dealt with during the experiment were two teams in which both members were near the same capability level. The worst-case team consisted of two *prima donna* programmers. The programming pair theoretically has equal responsibility for the team's efforts and product. We found teams functioned more smoothly, in spite of the members equally being

driver and navigator, if one member was slightly more capable than the other was. I read a statement by a software industry leader that stated hiring software engineers from the top 10 percentile of the top 10 universities would produce the best software development teams. I cannot imagine the stress that many egos can create on one project. Two strong egos of any caliber on a team create chaos until they recognize the power of two minds.

- **Common Area.** Coordination between the five teams would have improved if the teams had been working in a common area. Each team was located in a two-person cubicle, which limited the interaction between the teams. I use the term *war room* (or skunk works) to describe the ideal open environment, which would be a large area with worktables in the center and cubicles around the outside.

Some additional characteristics of the successful experiment are noteworthy. First, one of the manager's principle responsibilities was to buffer the teams from outside interference. The manager listed other important responsibilities that included referee (in the case of the prima donnas), arbitrator, coordinator, planner, cheerleader, and supplier of popcorn and other junk food.

Second, project managers must be supportive of the pair programming process. A classic (Theory X) manager observed a programming pair working on a design over a period of time. This manager suggested to their supervisor that one of the two programmers be laid off because only one was doing anything constructive. (The driver always gets the credit.) When the supervisor heard the suggestion, he replied that these programmers were the most productive people in the organization. The manager then asked that the programmers keep their office door closed so others would not get the same idea.

Summary

Most managers who have not experienced pair programming reject the idea without trial for one of two reasons. First, the concept appears redundant and wasteful of computing resources. Why would I want to use two programmers to do the work that one can do? How can I justify a 100 percent increase in person-hours to use this development approach? The project cannot afford to waste limited resources.

The second reason is the assumption that programmers prefer to work in isolation. Programmers, like most other people, have been trained to work alone. Yet according to the 1984 Coding War Games sponsored by the Atlantic Systems Guild, only one-third of a programmer's time is spent in isolation; two-thirds of the time is spent communicating with team members. Managers wonder about the necessary adjustments to another's work habits and programming style. They also worry about ego issues and disagreements about the product's implementation.

This experiment demonstrated strongly that programmers can work together effectively and efficiently to produce a quality product of which both programmers can be proud. Prior programming experience is not an issue. There are initial situations, especially with a team of equal experience and ego, where disagreements arise over who will be the driver. Those situations are generally transient. The benefits listed in the results section overwhelmed any personality issues that arose.

The second major benefit demonstrated in this experiment – a three order-of-magnitude improvement in error rate – is hard to ignore. Repairing defects after developments is much more expensive than uncovering and fixing the defects where they occur. The benefits of developing and delivering a stable product faster, reducing maintenance costs, and gaining customer satisfaction certainly minimize the risk of using pair-programming teams.◆

References

1. The Agile Alliance. "The Agile Manifesto." Software Development 9.8 (Aug. 2001).
2. DeMarco, Tom, and T. Lister. Peopleware. New York: Dorset House Publishers, 1977.
3. Weinberg, G. M. The Psychology of Computer Programming Silver Anniversary Edition. New York: Dorset House Publishers, 1998.
4. Beck, Kent. Extreme Programming Explained: Embracing Change. Reading, MA: Addison-Wesley, 2000.
5. Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. "Strengthening the Case for Pair Programming." IEEE Software 17.4 (July/Aug. 2000): 19-25.
6. Beck, Kent. "Embracing Change with Extreme Programming." Computer Oct. 1999: 71.
7. Hersey, P., and K. H. Blanchard. Management of Organizational Behavior, Utilizing Human Resources. Englewood Cliffs, NJ: Prentice-Hall, 1977.
8. McGregor, D. The Human Side of Enterprise. New York: McGraw-Hill, 1960.

Note

1. Theory X assumes the following: (1) Work is inherently distasteful to most people. (2) Most people are not ambitious, have little desire for responsibility, and prefer to be directed. (3) Most people have little capacity for creativity in solving organizational problems. (4) Most people must be closely controlled and often coerced to achieve organizational objectives.
2. Theory Y assumes the following: (1) Work is as natural as play, if conditions are favorable. (2) Self-control is often indispensable in achieving organization goals. (3) The capacity for creativity in solving organizational problems is widely distributed in the population. (4) People can be self-directed and creative at work if properly motivated.

About the Author

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, with more than 40 years of practical experience as a computer professional in hardware and software development. He developed the model that underlies the Sage and the GAI SEER-SEM software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He is currently preparing "Extreme Software Estimating" for Prentice-Hall, Inc. Dr. Jensen has a bachelor's of science degree in electrical engineering, a master's of science degree in electrical engineering, and a doctorate in electrical engineering from Utah State University.

Software Technology Support Center
7278 4th St.
Bldg. 100 G58
Hill AFB, UT 84056
Phone: (801) 775-5733
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil

Clarify the Mission: A Necessary Addition to the Joint Technical Architecture

Ingmar Ögren
Tofs Inc.

The Joint Technical Architecture (JTA) was published to provide the Department of Defense with the basis for needed seamless interoperability across its systems. The JTA contains three architectural views: operational, technical, and systems. The operational architecture view shows the tasks and activities for a system, while the other two views show supporting elements. It is important to understand how the three views relate to each other and how the parts of the technical architecture view support the overall system's missions and operators. This article discusses how a simplified and extended version of the Unified Modeling Language "component diagram" can be used to connect the three JTA views and consequently create a clarified environment for a system's software with a resulting increased probability that the right software will be built.

The Joint Technical Architecture (JTA) was created to establish a standard to provide interoperability among the Department of Defense (DoD) systems [1]. The JTA is a very ambitious and important document with great impact on creating and updating military systems in the United States. Over the long term, it also has considerable impact on civilian and foreign systems since its objectives and approaches go well beyond the U.S. defense sector. The objectives and main content of the JTA are best explained with a few citations from the standard itself:

- The JTA provides DoD systems with the basis for needed seamless interoperability.
- The JTA core contains the minimum set of JTA elements applicable to all DoD systems to support interoperability.
- The DoD JTA provides the minimum set of standards that, when implemented, facilitates this flow of information in support of the warfighter. The JTA standards promote the following:
  - A distributed information-processing environment in which applications are integrated.
  - Applications and data independent of hardware to achieve true integration.
  - Information-transfer capabilities to ensure seamless communications within and across diverse media.
  - Information in a common format with a common meaning.
  - Common human-computer interfaces for users, and effective means to protect the information.

The JTA defines three interrelated views within the architecture as shown in Figure 1: operational architecture (OA) view, technical architecture (TA) view, and systems architecture (SA) view.

> *"... as a software developer, it is not sufficient to understand the technical system; you must also understand the mission(s) and the expected and possible behavior of the operating roles ..."*

These three views are described in the following subsections (citation from the JTA). The article continues with the difficulties in consolidating these views.

The OA View

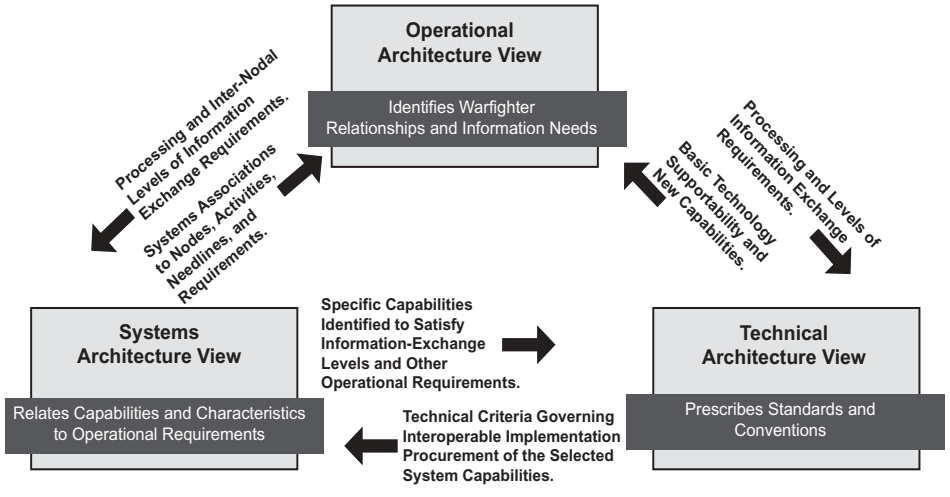
The OA view is a description of the tasks and activities, operational elements, and information flows required to accomplish or support a military operation. The OA contains descriptions (often graphical) of the operational elements, assigned tasks and activities, and information flows required to support the warfighter. The OA defines the types of information exchanged, the frequency of exchange, which tasks and activities are supported by the information exchanges, and the nature of information exchanges in detail sufficient to ascertain specific interoperability requirements.

The TA View

The TA view is the minimal set of rules governing the arrangement, interaction, and interdependence of system parts or elements, whose purpose is to ensure that a conformable system satisfies a specified set of requirements.

The TA view provides the technical systems-implementation guidelines upon which engineering specifications are

Figure 1: *Three Interrelated Views of Joint Technical Architecture*



based, common building blocks are established, and product lines are developed. The TA includes a collection of the technical standards, conventions, rules, and criteria organized into profile(s) that govern system services, interfaces, and relationships for particular systems architecture views, and that relate to particular operational views.

The SA View

The SA view is a description, including graphics, of systems and interconnections providing for, or supporting, warfighting functions. For a domain, the SA view shows how multiple systems link and interoperate, and may describe the internal construction and operations of particular systems within the architecture. For the individual system, the SA view includes the physical connection, location, and identification of key nodes (including materiel-item nodes), circuits, networks, warfighting platforms, etc., and specifies system and component performance parameters (e.g., mean time between failure, maintainability, and availability). The SA view associates physical resources and their performance attributes to the operational view and its requirements following standards defined in the technical architecture.

Inherent Problems with Interfacing

When you talk to anyone responsible for a complex system with multiple operators and a high content of software, you are likely told: "We concentrate on the mission, and our qualified people are our most important resource."

Modern defense systems include personnel (warfighters) to complete missions, with complex interaction between operator roles and technical system parts. This means that, as a software developer, it is not sufficient to understand the technical system; you must also understand the mission(s) and the expected and possible behavior of the operator roles, which are required to complete the mission(s) together with the technical system parts.

Furthermore, when you look at Figure 1 and the accompanying text, you see a multitude of relationships. However, they are rather informal and do not really tell you much about the common core behind the three views. Still further, when you look at the extensive definitions and standards for human-computer interfaces, they are lopsided, meaning that they limit their descriptions to the computer part of the interface and leave the human part to the reader's imagination.

The conclusion is that we have the following set of problems defined when working with the JTA in systems design:

- How do you really connect the three views to each other?
- How do you show how the elements of a system support the system's mission(s)?
- How do you clarify the human part of the human-computer interface?

Introducing Mission and Operator Objects

One way to solve these problems is to start from the Unified Modeling Language's[1] (UML) component diagram with its basic relationships: *depends on* and *included in*. Furthermore the diagram should be extended to include not only software objects but also mission objects, operator objects, and hardware objects. As a result you get a set of object categories as shown in the entity-relationship diagram[2] in Figure 2.

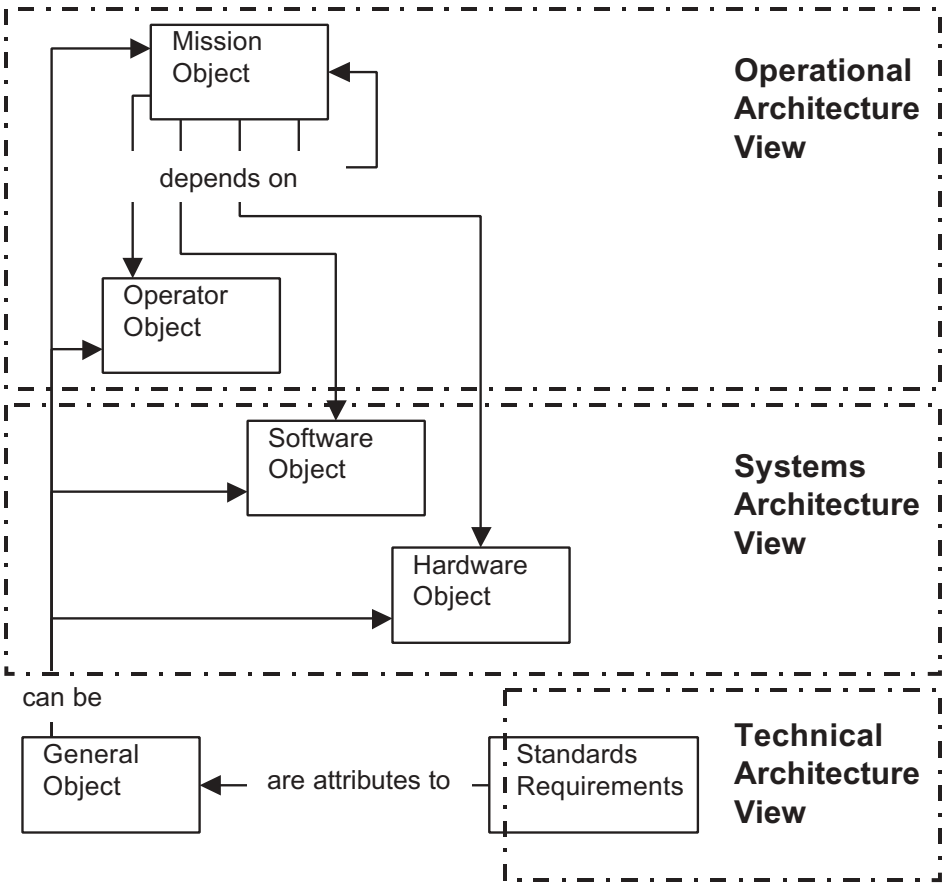
Figure 2 is a general entity-relationship diagram with objects and attributes to objects as entities drawn as boxes. Relationships are drawn as arrows and the diagram should be read along each arrow: <entity> <relation> <entity>. The diagram also applies the principle of *boxes within boxes* to show how smaller entities are parts of larger entities (the three JTA views). Note that the standards included in the TA view of the JTA concern all object categories through the *general object*, although they are primarily applicable to hardware objects.

This diagram shows one way to solve the problems identified above:

- How do you really connect the three views to each other? As shown in Figure 2, the elements in the OA view depend on the SA view, and all elements are derived from the general object. Consequently, the standard requirements of the TA view concern all objects as applicable.
- How do you show how the elements of a system support the system's mission(s)? *Mission objects* are introduced in the OA view to define missions on the system level and also on lower levels.
- How do you clarify the human part of the human-computer interface? Through introduction of operator objects, it is possible to define not only the computer side of human-machine interface, but also the human side with definition of the human operator's behavioral space to match the software's behavior.

The main message of Figure 2 is that

Figure 2: *Object Categories and JTA Views*



you can work with a general object concept that includes objects of categories: mission, operator, software, and hardware. To understand how a mission depends on system parts for completion, you use dependencies to define how each mission depends on other missions and on objects of other categories. This diagram also shows how this view of systems complies with the three views of the JTA.

Command and Control Example

Let us look at a hypothetical command and control (C2) example to illustrate the principles described above. (It is hypothetical since real systems are most often classified and too large for a short article.) While the JTA talks about command, control, computing, communication, intelligence, surveillance, and reconnaissance, this example will concentrate on the core mission, C2, without the support technology, which may or may not be included in a particular C2 system.

Figure 3 shows the top part (main missions) for a C2 system as four mission objects, drawn as a simplified UML component diagram. Note that the methods in the component diagram are represented by *abilities* required for the missions.

Figure 3 also shows how the component diagram can be compacted as a tree or as an indented list to show only the object types involved and their dependencies. This compact form, called the *Tree Graph*, can be used to show a somewhat more detailed view of the C2 system, as shown in Figure 4.

The Tree Graph is where you can see how the three aspects of the JTA can be managed together, clearly connected in the dependency tree as follows:

- The missions shown (C2 supported by plan tactical mission, train personnel, and develop tactics) together with the operator roles shown (planning officer, training officer, and tactics development officer) belong to the OA view in the JTA.
- The software system parts shown (planning support, training support, and tactics development) belong to the SA view in the JTA.
- The hardware parts shown (C2 computing system) are governed by the standards contained in the TA part of the JTA.

Here, the experienced reader may have the following objection: "You need much more to build a C2 system!" That is correct, and there are two reasons why so little of the C2 system is included in Figure 4. One is to limit the information to a reasonable amount for a journal article. The other is that this figure also concentrates on demonstrating how the Tree Graph can be used to show how a system depends on another system (visualizing *systems of systems*).

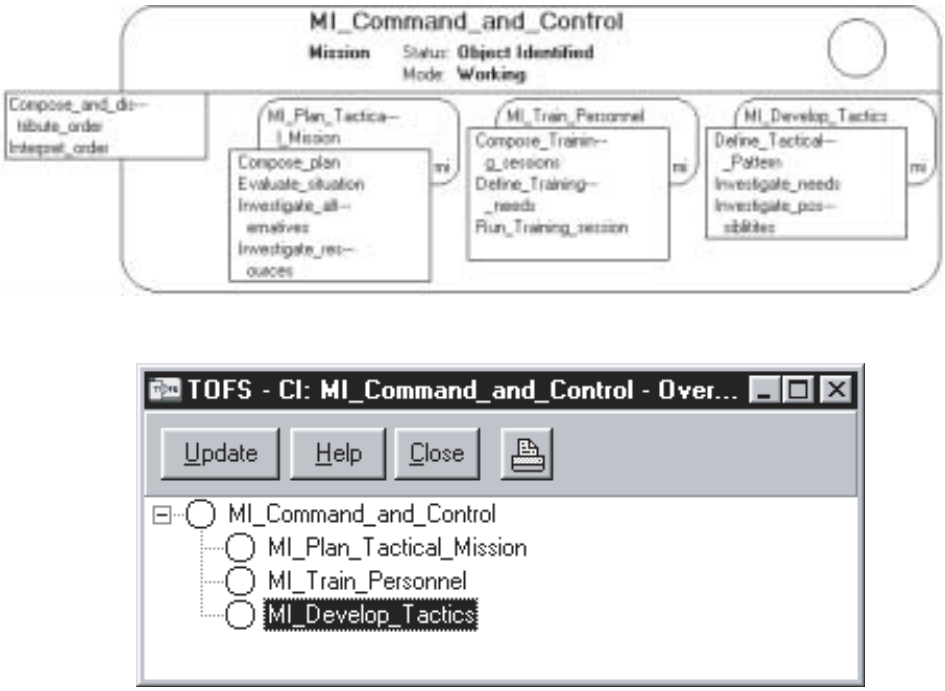


Figure 3: *Component Diagram and Tree Graph for the Top Missions of a C2 System*

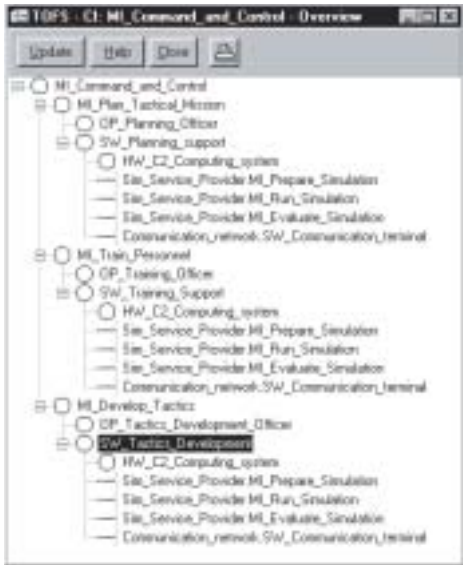
You can see how the three software objects shown depend on a communication network and on some external system to provide simulation services. This makes it possible to keep the C2 system and the simulation system separate and still clarify their interdependencies.

Practical Experiences

The principles related here have been applied in various real systems, primarily C2, communication, simulation, and avionics. The experience is that the principles work and result in system descriptions that are understandable both to end users and developers. However, since this is a new way of viewing a system, the following indications of uneasiness have been noted:

- Software engineers find it strange to work with the *human part* of human-computer interfaces. However, after some time they will most often accept that this is needed to build the necessary understanding to create the right software.
- Some end users might be afraid that working with the missions together with a contractor might result in *too much knowledge* of classified principles on doctrine with strategic and tactical principles. This may be a very real problem. Managing it requires an understanding that it is next to impossible for a contractor to produce a useful system without knowledge of why it is built, and how it should be used.
- Operator end users may be hesitant when contractors try to describe their work as system components (operator role objects). However, as soon as they really study these descriptions, they often get fascinated and produce some extremely valuable comments and criticism, which will be a real help to building the right system.
- People will consider it unnecessary to define the mission since "everyone

Figure 4: *Tree Graph for a C2 System*



knows what the mission is." This is correct, but when you start defining the mission, it is sometimes surprising how many different understandings you find within what *everyone knows*.

Perhaps the most important result of introducing mission objects, operator objects, and dependency trees is that they provide a common ground for end users and technical system developers to meet, which results in an increased probability of common understanding of the system.

Summary

The problems concerning the JTA's difficulties to consolidate its *three views* and to clarify system missions and operator roles in system architecture have been discussed. A possible solution based on a simplified and extended UML component diagram has been presented with a small C2 example. Furthermore, some experience from practical application of the principles presented has been described.

From the software engineer's point of view, this means that the software's environment is investigated and clarified prior to software design and programming. This will increase the software engineer's understanding of the software requirements and consequently also increase the probability that the right software is built.◆

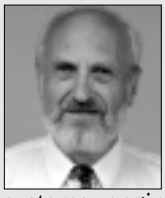
Reference

1. JTA Development Group. Joint Technical Architecture. Ver. 3.1. U.S. Department of Defense, 31 Mar. 2000 <www-jta.itsi.disa.mil>.

Notes

1. The following Web sites provide information on the UML component diagram: <www.sparxsystems.com.au/EAUserGuide/component_diagram.htm> and <http://jliusun.bradley.edu/~jiangbo/uml/Booch_uml/sld019.htm>.
2. The Tofs Web site provides information about the tool used for diagrams in this article <www.toolforsystems .com>.

About the Author



Ingmar Ögren has worked with the Swedish Defense Material Administration and various consulting companies in systems engineering tasks associated with communications, aircraft, and command and control. He is currently a partner and chairman of the board for Tofs Inc. and Romet, a systems engineering consulting company mainly utilizing the Objects for Systems development method. He also teaches systems and software engineering and has had several papers accepted at international conferences. Ögren is a member of Modeling and Simulation in Sweden and International Council of Systems Engineering. He has a master's of science in electronics from the Royal University of Technology in Stockholm.

Tofs AB
Fridhem 2
S-76040 Veddoe, Sweden
Phone: (+46) 176-54580
Fax: (+46) 176-54441
E-mail: iog@toolforsystems.com

LETTER TO THE EDITOR

Dear **CrossTalk** Editor,

In your December 2002 issue dedicated to engineers and scientists, two articles written by authors from the Air Force Materiel Command (AFMC) discussed the present plight of these occupations. Both articles discussed the problems in recruiting and retaining these skills for government service, especially for software engineering.

Also, within the last few months the secretary of the Air Force (SAF), and more specifically Lt. Gen. Stephen B. Plummer, principal deputy, Office of the Assistant SAF for Acquisition, has recognized the present and future problems of employing scientists and engineers. Plummer is principal deputy, Office of the Assistant Secretary of the Air Force for Acquisition, Washington, D.C., and the military director, U.S. Air Force Scientific Advisory Board.

However, in comparing the AFMC effort to the direction of SAF, there appears to be a conflict between the initiatives. The AFMC indicates a salary increase is needed, whereas the SAF reports survey results indicating that engineers do not consider pay to be a big issue.

Frankly, I wonder who participated in the survey the SAF referenced. Pay is a huge issue! In August 2000, the AFMC software organizations were surveyed to answer a question posed by Air Force Information Logistics: "Will the organic software functions be able to grow to meet the anticipated growth in software work?" The result of the survey at this Air Force base [Tinker] stated emphatically that salary is an issue that affects both hiring and retention.

At that time, the difference in salary between government and industry for comparable jobs ranged from 15 percent to 70 percent with government being lower. During the survey, we cited the Pay Comparability Act of 1990 and made the point very clearly that engineer and scientist pay should be in accordance with the law.

Shortly after this survey was performed, personnel in the information technology (IT) field, many of whom are in a nonprofessional job series and do not possess a college degree in computer science or engineering, received on average a 15 percent salary increase. In many cases this action caused IT personnel to be paid more than engineers, when their job responsibility was less. A request was made to include the software engineers into the IT salary action. No positive action occurred.

From that time until now, the situation has worsened. Our software organization has lost some of its better software engineering talent. Many of our former engineers have taken industry positions and have received significant pay raises. We foresee this to be an increasing trend if the government salary is not corrected to comparability with industry. In supporting two major weapon systems, there are partnering agreements between this software organization – the weapons manager, and his prime contractor for software support. The government software engineers work with contractor engineers to field common or integrated software products. It is more than a little distressing for government engineers to know what their counterparts are paid for equivalent responsibility.

Recently, we performed a study to determine the present salary difference between government and industry engineers. The results are tabulated in Table 1. As is seen, even with the severe economic downturn, the pay comparability situation is still very poor and appears to have worsened for personnel in management positions. In some cases, the government salary for software engineering management lags the average salary of industry by 70 percent to 80 percent.

Government software engineers deserve more than talk. Some amazing software capabilities have been created within the Air Force; two, specifically, are recognized as being world-class. The government functions compare favorably with anyone; they deserve to be paid commensurately. If an action is not implemented to overcome the pay differential between government and industry, it is safe to say these fine organizations will erode.

Here is something to think about: People organize to overcome unfair treatment by their employer. If pay disparity persists, government engineers and scientists are ripe for unionization.

Walt Lipke
Software Division
Tinker AFB, OK

Table 1: *Salary Comparison*

Salary Comparison Electrical and Electronics Engineers [1, 2]						
	NSPE			Government		
Equivalent Responsibility	10th Percentile	Average	90th Percentile	Step 1	Average	Step 10
NSPE – V = GS-12[3]	$58,441	$78,881	$97,010	$54,275	$62,069	$70,555
NSPE – VI = GS-13[4]	$69,583	$88,444	$112,300	$64,542	$78,753	$83,902
NSPE – VII = GS-14[5]	$72,150	$98,234	$133,500	$76,271	$96,608	$99,150
NSPE (non-(supervisory) = GS-13[6]	$53,292	$77,037	$93,260	$64,542	$78,753	$83,902
NSPE (5-9 prof supv) = GS-13[7]	$72,000	$96,103	$135,000	$64,542	$78,753	$83,902
NSPE (10-49 prof supv) = GS-13[7]	$79,550	$120,015	$169,000	$64,542	$78,753	$83,902
NSPE (≥50 prof supv) = GS-14 Branch Chiefs[8]	$92,185	$172,448	$293,800	$76,271	$96,608	$99,150
NSPE (≥50 prof supv) = GS-14 Division Deputy[9]	$92,185	$172,448	$293,800	$76,271	$96,608	$99,150
NSPE (≥50 prof supv) = GS-15 Division Chief [10]	$92,185	$172,448	$293,800			$115,633

Notes
1. This comparison utilizes information from the National Society of Professional Engineers 2002 Income and Salary Survey Report.
2. The general schedule pay scale and number of personnel within the Government Software Organization (GSO) are current as of 31 Oct. 02.
3. The GS-855-12 is the working level engineer in the GSO. Of the 397 GS-855s in the GSO, 25 are GS-855-12s.
4. The GSO employs 79 GS-855-13s consisting of approximately 1/3 supervisors, 1/3 team leads and 1/3 technical leads all utilizing the same pay scale.
5. The GS-855-14 position represents management at the branch level.
6. The GSO utilizes non-supervisor GS-855-13s as technical leads.
7. Most sections have a GS-855-13 section chief and team lead, and some GS-855-13 team leads are responsible for five to nine professionals supervised (prof supv).
8. There are seven GS-855-14 branch chief positions within the GSO.
9. The GS-855-14 deputy chief of the GSO manages half of the GSO, as well as standing in for the chief when necessary.
10. The GS-855-15 chief of the GSO manages 475 professionals, 531 total.

Let's Play 20 Questions: Tell Me About Your Organization's Quality Assurance and Testing

Gary E. Mogyorodi
Bloodworth Integrated Technology, Inc.

This article presents 20 questions used to determine and understand how mature the quality assurance and testing environments are within an organization. The questions are ordered to begin with those easiest to answer and become progressively more difficult for organizational compliance. An organization that has good quality assurance and testing practices tests early and tests often throughout its software development life cycle.

Currently, I am a senior consultant specializing in requirements-based testing. I have been in the computing industry since 1973. In 1993, I began specializing in testing and quality assurance. I have been teaching students how to write testable requirements, how to perform requirements-based testing, and how to find ambiguities in their requirements since 1998. Each time I teach a class, I find that I ask students the same questions about quality assurance and the testing environments within their organizations. Therefore, I have derived 20 questions that I use to try to understand how mature the quality assurance and testing environments are for each organization. After all, an organization that has good quality assurance and testing practices tests early and tests often throughout its software development life cycle.

I have ordered these questions so that, in my experience, the beginning questions are the easiest for an organization to comply with, and they become progressively harder to comply with as an organization proceeds through the questionnaire.

One point is awarded for each of the first four questions, two points for each of the next set of four questions, three points for the next set, and so on. In my experience, the last four questions are the hardest for organizations to comply with and are worth five points each. A perfect score is 60 points; I would expect very few organizations to score perfectly.

The questionnaire can act as a self-appraisal of the quality assurance and testing activities within an organization. A sample scorecard is included in this article.

The questionnaire does not cover every aspect of software development and testing.

Due to space constraints, CrossTalk *was not able to publish this article in its entirety. However, it can be viewed in this month's issue on our Web site at <www.stsc.hill.af.mil/crosstalk> along with back issues of* CrossTalk.

Did I Say "Koala Tea?"

Question: What do you call a formal afternoon reception or social gathering of furry, bear-like arboreal Australian marsupials, where they drink the boiled glossy leaves of an Asian evergreen shrub?

Answer: A "koala tea." Oh, wait – the issue this month was about quality, not koala tea. That's my mistake. I guess I really didn't know the correct definition of quality, which is "trait, characteristic or property, degree or grade of excellence[1]." But then, perhaps my first definition is just about as good as the second. It depends on the user's needs.

As usual with my columns, let me start with a story. Back in 1974, I was a young airman stationed at wonderful, beautiful Offutt Air Force Base. (Once you get on it, you can't get Offutt – the Air Force base with its own cemetery!) As an applications programmer at SAC (remember the Strategic Air Command?), a user asked me to sort a file of data for him. Being young and eager, I immediately set forth to write the ultimate sort routine to end all sort routines.

I had recently completed an undergraduate course in sorting and searching techniques, and I recalled a particularly cool sort from class (something like a combination polyphase-cascade-merge sort). It required several temporary files, but the machine we were using at the time was pretty limited. Does anyone remember the World Wide Military Command and Control System[2] (WWMCCS) Honeywell H6000? It had 96K of RAM. Not much memory space at all. However, it did have multiple tape drives, so I wrote a program that sorted using four tapes.

Of course with only five tape drives for the entire system, running my job took a bit of scheduling. In fact, it usually took a full day to actually run the job once I had submitted it. After a few weeks of development and testing, I was ready to unveil the program to my long-suffering user.

After explaining that there was a one-day delay between program submission and the results, I saw the look of dismay on his face. Come to find out, the data he wanted sorted consisted of exactly eight 10-character strings. A six-line bubble sort routine would (and eventually did) sort the data online in less than a second. The lesson I learned: Find out exactly what the user needs before writing the program. I had fallen into the trap of letting my biases and experience influence me into giving the user what I thought he needed, rather that giving him what he really wanted.

Well, I'm a bit older and wiser now (as my friends would point out, a lot older, maybe a little wiser) and frequently find myself teaching this lesson to others. It shouldn't be a secret; lots of people know about it. In fact, Simon and Garfunkel must have been software engineers. Back in the '70s, they produced their "Bridge Over Troubled Waters" album. If my memory serves me, on side two song No. 11 was titled "Keep the Customer Satisfied"(words and music by Paul Simon, 1970). Great song. Great message. "Just trying to keep my customers satisfied, satisfied."

You want a definition of quality? OK: Keep the customer satisfied. Do you know what customers want? Do you know what they really need? Are they part of the decision-making process when it comes to making trade-off decisions in the architecture and design? Are they part of the decision process when deciding what is part of a release, and what gets put off until the next release?

You see, you are not really ready to discuss quality until you know the users' needs. If their overwhelming requirement is accuracy, then speed might not impress them. If their overwhelming requirement is consistent uptime, then no matter how accurate and fast the system is, if it crashes frequently, they are not impressed. If their need is for a reliable system, and you forget to implement a quick-and-easy-to-use backup and recovery system, then they won't be impressed. Remember – the latest and greatest languages don't impress them if the resulting system is slow, unreliable, and buggy. A reliable Fortran program might just impress them more than a spiffy (but buggy) program written in Java Plus Plus Sharp (or whatever the language of the month is).

You want quality? Talk to your users. Find out what they need (and what they want). Involve them in trade-off decisions. Write the programs that will make them satisfied. Save the polyphase-cascade-merge sort for another time.

Oh – and yes – I do know that getting the real requirements out of some users is only slightly harder than communicating with the dead. I'll save that discussion for another column!

– David A. Cook
Software Technology Support Center/
Shim Enterprise, Inc.

1 All definitions are from Microsoft Encarta Encyclopedia, 1999 edition.

2. If you remember WWMCCS, then check out "Defense Department Classic Becomes Object of History" at <www.af.mil/news/Jul1997/n19970717_970866.html>.

FEELING A LITTLE TIED UP?

Why not get some *free* help?

ONE FREE CONSULTANT

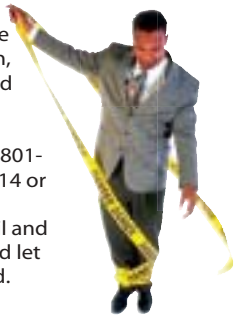
Get Some Help with Software Process Improvement

Good for assistance with configuration management, appraisals, personal and team software process, systems engineering, requirements, project management, software acquisition, Capability Maturity Model®, software quality and test, and process improvement.

ONE HOUR

That's right—one free consultant. If you are a manager, a practitioner, or a software engineering process group member who is committed to improving your software process, we can help you.

Software Technology Support Center (STSC) software process improvement (SPI) veterans can help answer questions and research your problems for up to an hour without charge to Department of Defense organizations. We can answer questions about starting SPI, key process areas, training, best practices, return on investment, and appraisals. And if our veterans can't produce an immediate solution, we will get you headed in the right direction.



Call the SPI Hotline at 801-777-7214 DSN 777-7214 or send us an e-mail at larry.smith4@hill.af.mil and mention this offer. And let us help you get untied.

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

Published by the Software Technology Support Center (STSC)

CrossTalk / MASE
7278 4th Street
Bldg. 100
Hill AFB, UT 84056-5205

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737