CrossTalk

July 2001 The Journal of Defense Software Engineering Vol. 14 No. 7

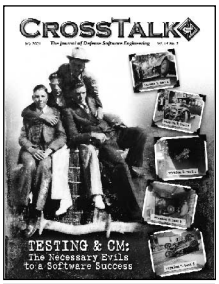
TESTING & CM:
The Necessary Evils to a Software Success

Testing & Configuration Management

Best Practices

Software Engineering Technology

Open Forum



On the Cover:
Kent Bingham, Digital Illustration and Design, is a self-taught graphic artist/designer who freelances print and Web design projects.

Main photo reference courtesy Cathye Cox, Hooper, Utah. Car crash photos courtesy Western History/Genealogy Department, Denver Public Library.

Departments

CrossTalk

Sponsor	*Lt. Col. Glenn A. Palmer*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pam Bowers*
Associate Editor	*Benjamin Facer*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Jensen*
Phone	(801) 586-0095
Fax	(801) 777-5633
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk/crostalk.html
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. 18.

Ogden ALC/TISE
7278 Fourth St.
Hill AFB, Utah 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.

The Unsung Heroes of Testing and CM

Lack of configuration management (CM) processes can really hurt! I remember one experience from the days when my normal development cycle was iterations of hacking and then seeing if that function worked. I was developing software that interacted with a CM database, when the software suddenly stopped working. I went to my previous version to try to discover which change had caused the software to stop working, but my previous version wasn't working either. After two hours of searching and finding no clue why suddenly nothing worked, I checked the CM database. I discovered that all of the previous data had been dumped and replaced with different data. When I asked the project lead about the change, I was told that she had changed all the database contents. When I explained that not knowing this information had caused me to spend two hours searching for the cause of a non-existent software problem, the simple response was, "Better you than me."

The glory, fun, and excitement in software engineering are the design, development, and building of something tangible (software). Software engineers who design are the all-stars, heroes, and white-collar hotshots of the industry. However, behind all that glitz and glory, someone has to keep track of all the details (CM) and verify that product quality is sufficient for the client (testing).

Although these tasks require the same knowledge and experience as a designer, software engineers that work CM and testing are sometimes looked upon as less glamorous engineers. In reality, they act as the glue that holds best practices in place. Many problems that occur in software development can be traced back to poor CM and testing procedures. So they are necessary to get the software developed but akin to evil because the devil is in the details.

While developing code in a Level 5 organization, I found that CM processes and peer reviews caught more errors before testing than any other processes. Then I knew I could count on our test engineer to catch remaining errors. This month's article *Software Configuration Management: A Discipline with Added Value* shares the CM process used at Hill Air Force Base's Software Engineering Division. Hopefully, CROSSTALK readers will see that CM can do more than just provide a library for old code and documentation. Next Robert McCann discusses the tradeoffs in time and value for code inspection vs. testing in *How Much Code Inspection Is Enough?* In *The Problem with Testing,* Norman Hines reminds us that while testing is important, we need to rely on previous stages of the software development life cycle to ensure a quality end product.

The United States Air Force's (USAF's) Capability Maturity Model® support has resulted in many USAF organizations being at least Level 3. I have been very happy with my career since joining the USAF due to the atmosphere fostered by an organization focused on process improvement. CROSSTALK'S Publisher Tracy Stauder shares some additional advantages of working for the USAF in *A Smart Way to Begin a Civilian Engineering Career in the U.S. Air Force.*

In this month's other supporting articles, Dr. Randall Jensen offers ideas on improving the capabilities and dependability of estimation tools in *Software Estimating Model Calibration.* Lt. Col. Robert Lang shares software maturity information from the Defense Contract Management Agency in *Process Capability Data for the Asking.* Finally, Lawrence Bernstein and David Klappholz bring software development reality into the classroom with *Getting Software Engineering into Our Guts.*

My experience has proved that working with good CM and testing processes is much better than working in a CMM Level 1 environment. When working in a Level 5 organization, I had better work hours, a better collaborative environment, and better confidence in my work because of the better end product. The people performing the CM and testing functions are a critical part of the team committed to an end goal of outstanding quality that encompasses the spirit of a Level 5 organization.

Elizabeth Starrett
Associate Publisher

Software Configuration Management: A Discipline with Added Value

Tresa Butler, Verla Standley, Elaine Sullivan
Ogden Air Logistics Center, Technology and Industrial Support

Faith Turner
Scientech, Inc.

From the beginning of our software engineering organization to our current Capability Maturity Model® (CMM®)Level 5 quality practices, the implementation of the software configuration management (SCM) discipline combined with management and engineering practices has been critical to our weapon-system software sustainment activities. The focus of this article is to discuss how SCM adds value to our organization by establishing and maintaining continuity of the engineering workflow and provides information to help establish a strong SCM function in maturing software organizations.

The word "Kaizen" is a Japanese term used to define the discipline of continuous improvement. For example when an automotive assembly line is considered to be perfect, it is pushed past its limits until a malfunction occurs; the anomaly is found and corrected, and then the limits are tested again. When applied in the workplace, Kaizen means continuing improvement involving everyone – managers and workers alike.

Software configuration management[1] (SCM) is the one discipline where development, sustainment, support, and software Kaizen are accomplished to achieve quality products. SCM defines, implements, and manages product life cycles by planning, identifying, controlling, auditing, and improving the elements by which they are created.

During the early years of our Software Engineering Division (TIS) at Hill Air Force Base, Utah, SCM was not a term commonly used. Most engineers were aware of SCM, but would prefer to ignore it. SCM meant processes and procedures. The engineers were there to write software and did not want to be bothered with process and paperwork. Every individual or team had their own way of doing things, resulting in little work uniformity. They did, however, want to have quality software with as little rework as possible.

This desire to provide quality software forced us to look at how we did business and to search for ways to improve. Management chose to use the Software Engineering Institute's Capability Maturity Model® (CMM®), and after an initial review began building toward process improvement. This decision really introduced SCM as a key process for better software to everyone within the division. In order to become a CMM Level 2 organization we needed to have consistent policies for managing our software projects. The standards set by the CMM for this level were requirements management, software project planning, software project tracking and oversight, software subcontract management, software quality assurance, and SCM.

During our CMM implementation, software quality assurance became tied to SCM and evolved as a major player within the structure of the organization. This was not an easy road for the SCM team. Many people within the division did not want the change and did not want the extra demands that SCM would put on them. It meant being accountable for every line of code as well as documenting every change.

Until this point, each engineer was accountable for his or her own configuration management. Now it was necessary to have a separate group of individuals with the sole purpose of providing quality assurance as well as managing the elements of each project injected into the processes.

It meant following a process that was rigid enough to keep everyone on the same track, but flexibile enough to allow each team to develop its products based on customer demands. Over time, SCM has become the discipline that assures quality software. In short, SCM has become the *glue* to an organization that produces quality products.

Developing the Glue

During the developmental stages of our weapon-system software activities, SCM plays a major role in planning and managing the schedules and milestones used during the project life cycles, as well as identifying product configuration items (CIs). Within TIS, SCM defines and records the origin and details involved in the inception of the product by establishing baselines. When SCM disciplines are used during this initial developmental stage of the product life cycle, it is comparable to the parable of the man who built his house upon the rock. A solid SCM discipline provides a firm foundation upon which software development and sustainment are achieved.

It is during the sustainment stage of weapon-system software activities that the SCM discipline provides consistency and strength. It is no longer adequate to simply create a product using a set of established ground rules and guidelines; now a structured enforcement of processes is a must. SCM provides continuity to the workflow by establishing the processes and procedures for controlling and auditing CI's throughout the product life cycle to ensure quality, integrity, and accountability levels are met and maintained.

SCM plays an integral part in scheduling, attending, and recording pertinent information during the definition portion of the project. SCM enhances the sustainment stage of the product by carefully tracking each software activity, thus blending in integrity and quality through repeatable auditing and data control. Establishing traceable metrics to track costs, identify weaknesses, and determine recovery capabilities ensures SCM as a value-added entity to the product life cycle of our organization. It assures that every requirement, problem, action item, etc., is tracked to closure, and that metrics data for each of these activities are updated.

® The Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.

SCM maintains a configuration status accounting (CSA) record of requirements compliance, cost control, source lines of code, and more. This data is used in information exchanges such as program management reviews to make well thought-out, informed decisions. SCM's data management of workflow, as well as maintenance of product life cycles, provides sustainment for past, present, and future projects.

The Life Cycle Workflow

Within our organization, the SCM function defines, implements, and manages product life cycles by planning, identifying, controlling, and auditing the current workflow. Having a well-defined process has enabled us to adapt new hardware and software workloads into our organizational workflow.

One thing that has proven to be very beneficial to our organization is the ability to tailor our formal configuration management (CM) process to the needs of each individual project. This way each individual team does not have to adhere to strict procedures; instead, each team is allowed flexibility within their own programs. SCM has been very helpful to each team in setting up procedures that comply with the process, but also fit individual needs. Following is an outline of that workflow process:

Planning: Management utilizes SCM to establish and maintain CM and project plans that define project activities and deliverable work products. This includes processes and procedures for the life span of the project. SCM is used to attend and record project directives, schedules, data requirements, peer reviews, and configuration control boards (CCB). These boards define milestones, deliverable work products, and cost and schedule.

Within our organization, the CCB is held prior to initiation of any work to define requirements, schedules, and deliverables, which are incorporated into a project directive and project requirements document. These signed documents represent the agreement between our organization and our customer defining project milestones and deliverables. CM configures these documents for referral throughout the life of the project, and these documents are reviewed periodically for additions/deletions.

```
                    _P_R_O_J_E_C_T___R_E_P_O_R_T_

TV3.2  CANDIDATES
Tue Apr 10 12:11:08 2001
-------------------------------------------------------------------------------
Project: #C1-R0     Parse Configuration and BE Data
SYS DES:                                                        Status: OPEN
INT TEST:                                SYS:    0 Mhrs      ICR:    0  Mhrs

SUB     DESIGN ENGINEER     HRS  SOFTWARE ENG         HRS  SLOC  ERT ENGINEER
---     ---------------     ---  ------------        ---  ----  ------------
AFMSS                            STARRETT, BETH       13    50  F Montierth
-------------------------------------------------------------------------------
Project: #C2-R0     Time-Line Data
SYS DES:                                                        Status: OPEN
INT TEST:                                SYS:    0 Mhrs      ICR:    0  Mhrs

SUB     DESIGN ENGINEER     HRS  SOFTWARE ENG         HRS  SLOC  ERT ENGINEER
---     ---------------     ---  ------------        ---  ----  ------------
AFMSS   RICHARDSON, JIM      24  RICHARDSON, JIM      48   280  F Montierth
-------------------------------------------------------------------------------
Project: #C3-R0     Export TableView Data
SYS DES:                                                        Status: OPEN
INT TEST:                                SYS:    0 Mhrs      ICR:    0  Mhrs

SUB     DESIGN ENGINEER     HRS  SOFTWARE ENG         HRS  SLOC  ERT ENGINEER
---     ---------------     ---  ------------        ---  ----  ------------
AFMSS                            STARRETT, BETH       14   160  F Montierth
-------------------------------------------------------------------------------
Project: #C4-R0     Allow Multiple Lists in Table View Data
SYS DES:                                                        Status: OPEN
INT TEST:                                SYS:    0 Mhrs      ICR:    0  Mhrs

SUB     DESIGN ENGINEER     HRS  SOFTWARE ENG         HRS  SLOC  ERT ENGINEER
---     ---------------     ---  ------------        ---  ----  ------------
AFMSS   RAISOR, KENNETH       3  RAISOR, KENNETH      12  1300  F Montierth
-------------------------------------------------------------------------------
Project: #C5-R0     Allow For All File Extensions
SYS DES:                                                        Status: CLOSED
INT TEST:                                SYS:    0 Mhrs      ICR:    0  Mhrs
```

Figure 1: *Sample Project Report*

Identifying: Upon completion of the upper level planning, the CSA database is populated by SCM to begin the task of identifying each configuration item and to begin gathering metrics for the life-cycle updates. By obtaining metrics for proposed work products, SCM provides management and engineering with the necessary data to make judicious decisions regarding weapon-system software sustainment activities. This is the heart of the continuous process improvement of our Level 5 organization.

The SCM team works with the program managers to identify the project's CIs. SCM populates the tracking databases by creating work products and their related data management objects and ensures that all requirements approved during the planning state are incorporated into the update. This requires creation, maintenance, and closure of work products for schedules, engineering change proposals, system design change requests, subsystem design change requests, software change requests, source lines of code, etc. The database then provides pertinent information for accumulating proposed weapon-systems upgrades.

Our organization uses our CSA system to record many types of metrics as well as actual man-hours and lines of code dedicated to each change request produced. These actual metrics are later used when accessing assets and manpower for new workloads. Figure 1 is a sample project report that includes the project identifier, project name, engineer assigned to the project, man-hours, lines of code, and other information that may be required during project development. There are several other reports that can be pulled to show the status of projects, percent complete, and other valuable metrics.

Controlling: SCM enforces control of CIs by establishing processes and procedures to maintain accountability of configured software enhancements throughout the life cycle of the upgrade. Incremental configuration at each stage/milestone ensures incorporation of approved source code and maintains traceability of known anomalies. Within our organization, the CM functions to update the CSA database at intervals during the product life-cycle, providing a current snapshot of the program at any given time. This means that when addressing both current and archived projects, the historical data regarding incremental releases describes during which phase anomalies were identified, along with in which release the anomalies were corrected.

Figure 2 (see page 6) is an example of a Software Change Request (SCR) form as it is recorded within our database. Within this process several metrics are recorded for future use. Dates are tracked as each milestone is passed such as the completion of a final peer review and the approval of the CCB, as well as the man-hours spent reviewing SCRs, time spent in peer

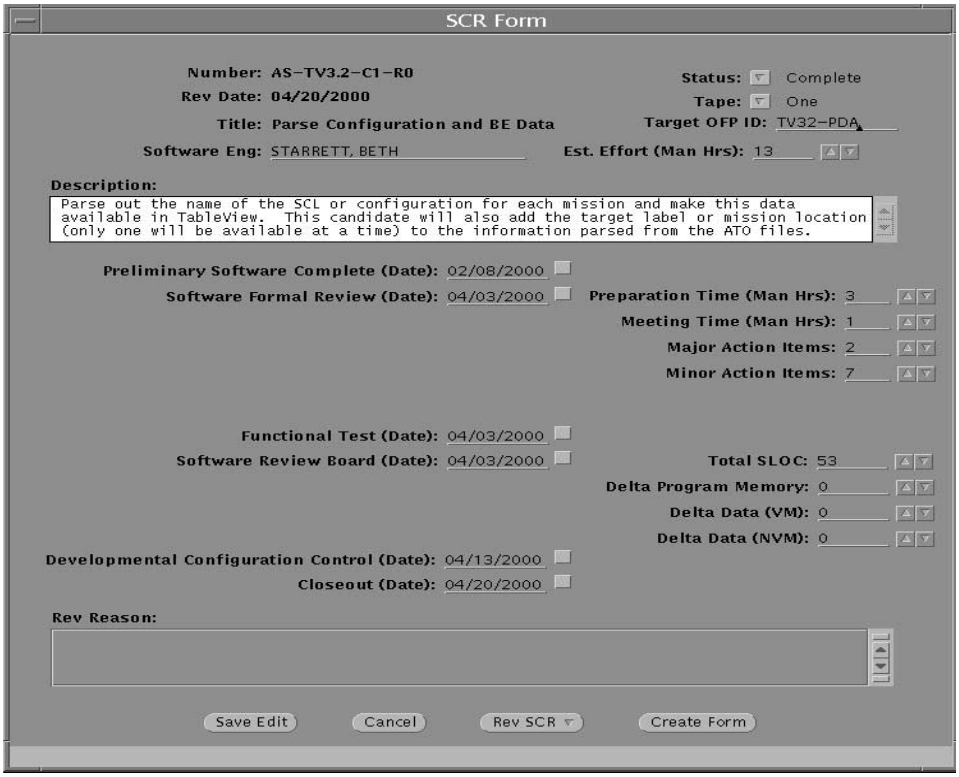


Figure 2: *Software Change Request (SCR) Form*

reviews, number of action items, their priority, source lines of code, and memory changes.

Anomalies are found during various testing phases. As they are detected the engineer responsible for finding the error enters them into the database. The database assigns the anomaly a tracking number after which the anomaly becomes a configured item. SCM tracks the error until it is fixed or determined not to be an error. Figure 3 is an example of a report used to show the status of anomalies.

Auditing: SCM produces audits at incremental steps throughout the software-building process to ensure quality, integrity, and adherence to established processes and procedures. Additionally, SCM incorporates quality assurance throughout the life cycle of the product by being a separate entity and maintaining continuity and accountability of the engineering workflow. Our CM processes include audits and quality checks ensuring that specifications are being updated incrementally as software lines of code are being developed.

An example of these audits within our organization occurs after a change request has been reviewed, comments recorded, and the author has accomplished a re-edit to include peer review comments. Our peer review process requires that an audit of the document be performed to ensure that the author incorporated all the approved changes. If the audit is not passed, the change request repeats this step of the process.

One of our lessons learned occurred when a customer provided us with documentation that required an upgrade prior to releasing new software. The customer requirements were vague and undefined and did not include all CI and identifiers needed. Additional hardware modification requirements were not identified until after we began upgrading the software. Identifying, correcting, and implementing the anomaly at such a late date impacted the software release cycle. To prevent this anomaly from reoccurring, a corrective action to tailor SCM processes resulted in a preliminary review of all documentation. This has eliminated 90 percent of the problems we had previously encountered. This resulted in the CCB receiving a better product to review and more accurate estimates of program cost/schedule.

Tools and Metrics

Tools are one of the key capabilities within our software sustainment environment. Tools provide the identification and control of the software and its related components as they change during the software's life cycle. There are numerous off-the-shelf SCM tools available in today's market, some of which we use in our organization. But we have developed many organic tools to comply or adapt specifically to our processes and corporate culture. There are traditional CM tools that provide check-in/check-out control of code as well as the ability to compile or build. Within our organization we have tools that provide process management such as the ability to track anomalies and provide problem

Figure 3: *Anomoly Status Report*

Text Editor – TV3.2_SPAR_Summary_ALL.rpt

File Edit Format Options Help

_S_U_M_M_A_R_Y___S_P_A_R___R_E_P_O_R_T_

Baseline: TV3.2
Group: ALL
Wed Apr 11 10:37:24 2001

SPAR	Grp	Pri	Status	Class	Title
1	AFMSS	LOW	Deferred	U	AOI disabled when map is not available
2	AFMSS	MED	Fixed	U	ICDs Not Updated for C5
3	AFMSS	LOW	Cancelled	U	SDD File List Tables Outdated
4	AFMSS	LOW	Fixed	U	Table View Shape Column
5	AFMSS	MED	Cancelled	U	GUI ICD References Non-Existent Laydown Functions
6	AFMSS	LOW	Fixed	U	Can't build Editor for PFPS 3.01
7	AFMSS	HI	Fixed	U	The Check Laydown Function does not work.
8	AFMSS	HI	Fixed	U	ACO Set ID GEOLINE Is Not Being Parsed.
9	AFMSS	MED	Fixed	U	Installation Auto Selection fails with PFPS 3.1.1
10	AFMSS	HI	Fixed	U	Laydown Editor Fails to Launch
11	AFMSS	HI	Fixed	U	Can't Compile Laydown Editor
12	AFMSS	MED	Fixed	U	Laydown Editor can't open some files.
13	AFMSS	MED	Fixed	U	Can't parse all formats of EFFLEVEL
14	AFMSS	MED	Fixed	U	Confirm Build Mission Dialog needs scrolled list
15	AFMSS	MED	Fixed	U	Extra Square Box in Table View
16	AFMSS	HI	Fixed	U	Points may not be sorted in the proper order
17	AFMSS	HI	Fixed	U	TableView ACO List Errors
18	AFMSS	HI	Fixed	U	Text button displays wrong info in TableView
19	AFMSS	HI	Fixed	U	TableView Column Headers
20	AFMSS	HI	Fixed	U	Target Columns do not show all points
21	AFMSS	HI	Fixed	U	Customize TableView Problems
22	AFMSS	LOW	Deferred	U	FalconView Covers TaskView Window On

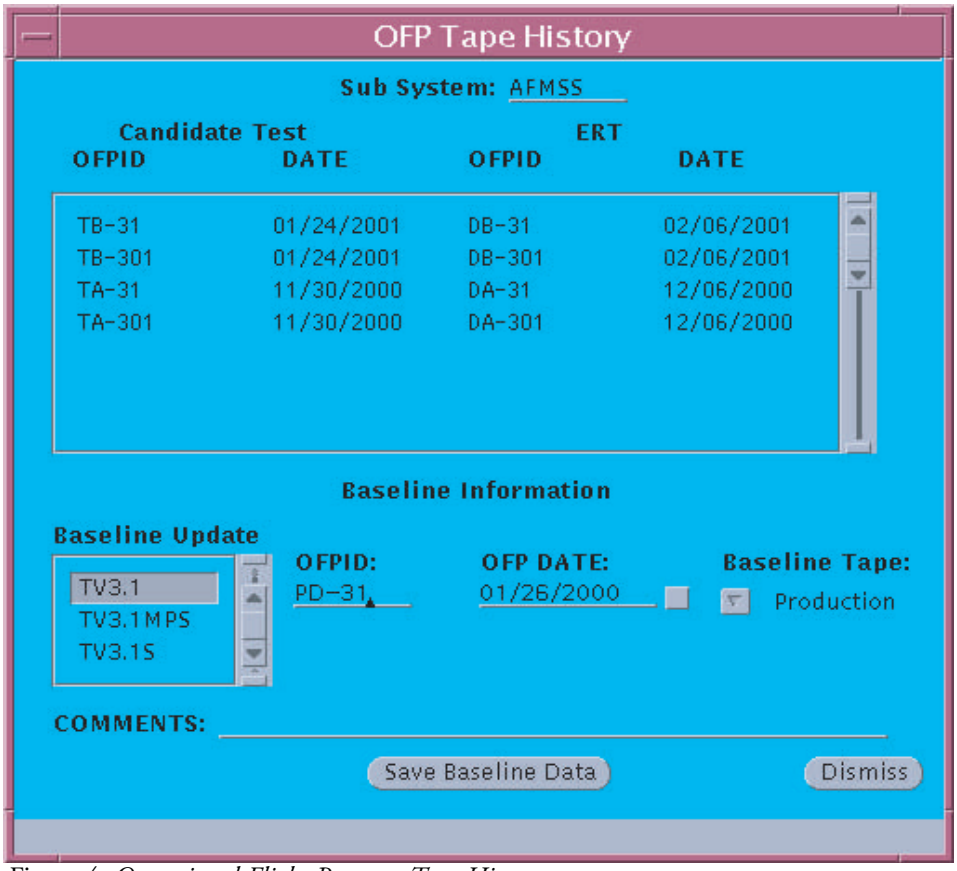


Figure 4: *Operational Flight Program Tape History*

reports. We also have commercial off-the-shelf tools that have been tailored to fit into our process and provide us with the necessary quality checks and balances.

One of the key areas within SCM is status accounting. To have true CM, tools are required to track the status of all CIs as well as problems or anomalies. We have developed tools within our organization that provide the ability to track several complex systems as well as to collect data and generate numerous reports. These tools also provide versatility in adapting to a variety of different workloads. One of the advantages of an organic tool is the ability to adapt or to change the tool as new requirements come in or as processes are updated.

An example of a new requirement added to our database recently is the Tape History dialog box shown in Figure 4. This history helps us to track the baseline used to develop the current update of the Operational Flight Program (OFP). It tracks all version releases of the software identified by the OFP Identification Number and the dates they were compiled. As a result this has helped us to track baselines for individual subsystems and enables us to pull reports for those who need this information.

Throughout the different stages of process improvement, from being a CMM Level 1 to a CMM Level 5 organization, we have learned many valuable lessons. One of the most valuable being the importance of creating your processes and then buying or developing a tool that compliments that process. If you buy a tool without knowing where you are going and what your overall goals are, you end up having to adapt your process to fit the tool. That may not be the best for your organization.

One of the benefits of a good status accounting tool is the metrics and reports that can be obtained. This has been very beneficial to our organization and to our customers. We are able to track estimated costs, man-hours, source lines of code, anomalies, memory requirements, rework, and much more. This gives us the ability to compare estimated data to actual data. It is this historical information that gives us a solid track record and helps us greatly in bidding on future workload, as well as supplying our customers with information in creating project requirements.

Training

Training for each member of the SCM team is a must. SCM is an evolving field. Not only is it necessary to keep up with new ideas and tools, but to keep up with the industry on how SCM is being interpreted by the world or even in other parts of our own division. In order to provide customers with current processes and procedures, we must be aware of changes and improvements made within the industry. This enables assigning appropriate authorities and responsibilities to all SCM activities within our organizations.

Management, engineers, and configuration managers must understand the processes within their respective organizations. It is necessary to be knowledgeable enough to tailor SCM practices to the needs of each customer/workload. Configuration managers are involved in all stages of development; they must become an integral part of the process. Here, they are depended on for their expertise in decision making to facilitate process improvement and provide a quality assurance role. Detailed knowledge, formal training, and on-the-job experience result in the ability to recognize problems – to stop work, address issues, correct problems, and continue moving ahead. When problems are encountered, knowledgeable configuration managers are invaluable in resolving issues. Some training examples that benefit our organization follow:

- First is mentorship between trained pesonnel and new/untrained personnel.
- Second is CM training courses. These courses give a broad overview of SCM and usually benefit anyone interested in becoming knowledgeable of basic SCM fundamentals.
- Third, and most importantly, is on-the-job training, which provides the most insight to SCM processes and procedures.

Properly trained SCM personnel result in procedures that produce repeatable quality products. Members of the SCM team are not technical or engineering people. The team is comprised of individuals who are competent in the skills needed to provide insight and background to SCM policies and procedures. Within our organization, this has provided an avenue for individuals to pursue the

set criteria for developing configuration management skills, which results in opportunities for advancement. Proper training creates knowledgeable configuration managers who can teach others the value of the SCM discipline. Trained configuration managers perform their duties with confidence and professionalism. These software professionals make the SCM discipline a vital function in maturing a software organization.

Conclusion

Maintaining a SCM discipline is critical to our CMM Level 5 software sustainment activities. Proper implementation of SCM enables us to plan, identify, control, and audit product life cycles. SCM along with management and engineering guide our organization to continuously improve our ability to meet expectations of high quality, low cost, and on time deliveries. Continuous improvement, or Kaizen, can be achieved when practitioners are provided with proper tools, adequate training, and empowered with a quality process.◆

Note

1. Configuration management definition is as defined by the Configuration Management Training Foundation (CMTF), Magalia, Calif.

About the Authors

Tresa Butler is a software configuration manager at the Ogden Air Logistic Center, Software Engineering Division at Hill Air Force Base, Utah. She has worked for the Department of Defense for 13 years with the past nine years in data management and software configuration management. She participated in the software configuration team that was assessed by the Software Engineering Institute as a Capability Maturity Model Level 5. She is currently the Software Configuration Management lead for F-16 Operational Flight Programming workload. Butler attended Weber State University and plans to continue her education in the fall.

6137 Wardleigh Road
Hill AFB, UT 84056-5843
Phone: (801) 777-6809
DSN: 777-6809
E-mail: tresa.butler@hill.af.mil

Faith Turner is a software configuration manager contracted through the F-16 (SPO) at Ogden Air Logistics Center in Utah, to provide configuration management support for software development at Hill Air Force Base (HAFB). She played an integral role on the software configuration management team during a Software Engineering Institute assessment that resulted in the first Capability Maturity Model Level 5 rating at a government organization. Turner has 16 years experience in the configuration status accounting and configuration management field. She has been a member of the F-16 software configuration management team for six and one-half years. Turner attended Texas Women's University and is continuing her education at Park College, HAFB.

Scientech, Inc.
6137 Wardleigh Road
Hill AFB, UT 84056-5843
Phone: (801) 775-3104
DSN: 775-3104
E-mail: faith.turner@hill.af.mil

Verla Standley is a software configuration manager for the Automatic Test Equipment (ATE) in the Software Engineering Division at Hill Air Force Base. She has worked for the government for 22 years and has been a configuration manager for 10 years. Standley was a member of the software configuration team that was assessed by the Software Engineering Institute as a Capability Maturity Model Level 5. For the past three and one-half years she has been the Software Configuration Management lead over the ATE workload. Verla attended Weber State College and has taken several configuration management classes.

7278 4th Street
Hill AFB, UT 84056-5205
Phone: (801) 777-0960
DSN: 777-0960
E-mail: verla.standley@hill.af.mil

Elaine Sullivan is a software configuration manager in the Ogden Air Logistics Center, Software Engineering Division at Hill Air Force Base, Utah. She has been involved with configuration of F-16 software since 1988 and is currently the Software Configuration Management lead over the Avionics Intermediate Shop Workload. Sullivan developed and implemented the configuration process for her area and was instrumental in writing the configuration processes for the division and branch. She was an integral part of the software configuration team during a Software Engineering Institute assessment that resulted in the first Capability Maturity Model Level 5 rating at a government organization. She has an associate's degree from Ricks College, Rexburg, Idaho.

6137 Wardleigh Road
Hill AFB, UT 84056-5843
Phone: (801) 775-2878
DSN: 775-2878
E-mail: elaine.sullivan@hill.af.mil

"If I had to sum up in one word what makes a good manager, I'd say decisiveness. You can use the fanciest computers to gather numbers, but in the end you have to set a timetable and act."

— Lee Iacocca

How Much Code Inspection Is Enough?

Robert T. McCann
Lockheed Martin Management and Data Systems

Given code inspection effectiveness (defects found during inspection/defects found during inspection plus those found during test) as a function of preparation rate (amount of code examined per labor hour), it is then possible to construct a simple cost model that predicts testing labor hours as a function of code inspection preparation rate. This paper develops that model and computes the optimum code inspection preparation rate to minimize total cost (inspection + test). Existing program data (with significant caveats) have been used together with certain rough approximations to show that Fagan-style code inspections obey a simple predictive cost model.

The purpose of this paper is not to present actual performance data but to demonstrate how such data can be analyzed in the context of a cost model extending over multiple development processes: code inspection, code inspection rework, test, test rework, and regression testing.

The purpose of the model is to demonstrate how to reduce development costs by managing the amount of time spent preparing for software inspections. All program data have been modified to protect the proprietary nature of that data. The conclusions and basic nature of the model are unaffected by these modifications.

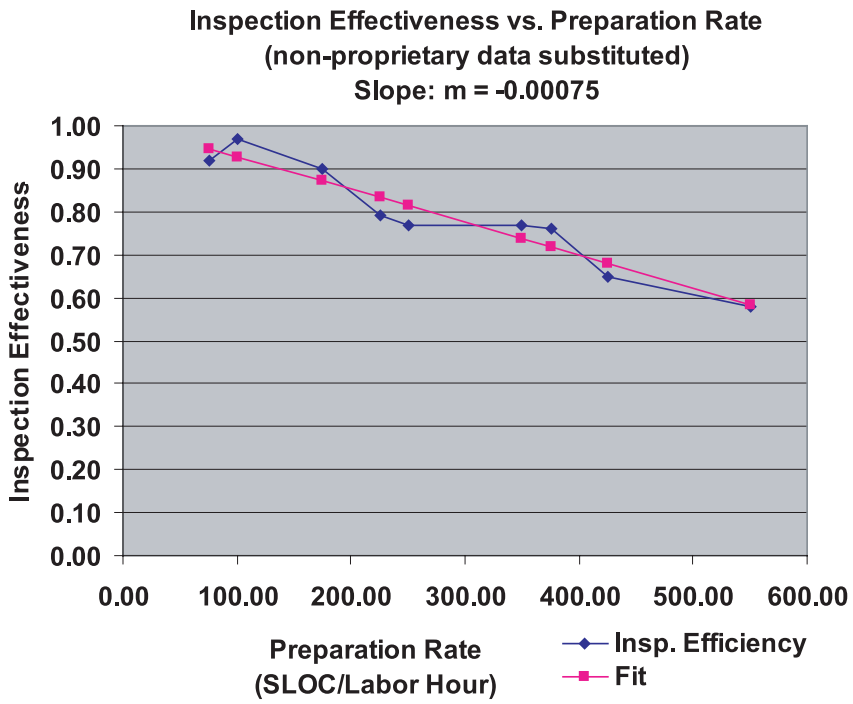
This model can be used to optimize performance of future programs using a similar Fagan-style [1, 2] code inspection process with significant cost and schedule savings as well as quality improvement – in effect, better, faster, and cheaper. Such performance improvements should result in improvements in profitability, competitiveness, and customer confidence in both contract performance and product quality.

Further, the model is easily modified to other work products and cost drivers, e.g., design complexity. Given that code inspection may not be particularly effective in finding design and requirements defects, it may make sense to extend the model to include more development processes, such as requirements development and requirements inspection, design development and design inspection, defect repair and associated inspection, etc.

Inspection, Test, and Accounting Processes

To perform this kind of analysis, it is necessary for a project to have defined, consistently executed processes for code inspection, code inspection rework, testing, and testing rework that collect and retain sufficient data. It is especially important in deriving this model that the inspection process was nearly statistically stable with respect to inspection preparation rate (lines of code reviewed per labor hour). Otherwise, the statistical fits would have little predictive value. The cost data are also assumed available and traceable to these processes.

Figure 1 : *Least Squares Fit of Inspection Effectiveness*



Inspection Effectiveness

It is not possible to know the true, exact percentage of code defects found by code inspections until all defects have been discovered; furthermore that is not likely ever fully to happen. However, if it is assumed that a significant fraction of major code defects found in testing could have been found in code inspection, then there is value in studying the code inspection effectiveness.

This is the ratio of major code inspection defects[1] divided by the sum of major code inspection defects plus the major code defects found by testing. The code inspection effectiveness ratio has the desirable property of ranging from zero to one even though the relationship between major inspection defects and test defects is many-to-many rather than one-to-one. The model has the further benefit of quantifying how inspection preparation regulates testing costs.

When code inspection effectiveness is averaged by computer software configuration item (CSCI[2]), it is possible to develop a correlation between code inspection preparation rate and code inspection effectiveness. The correlation is well fit by a straight line with a y-intercept of 1.0 (see Figure 1). The F-test [3] shows a confidence level in excess of 99 percent with the linear fit accounting for more than 90 per-

cent of the variation in the data. The F-test is a statistical test to verify that the function used to fit the data did not do so by accident.

Please note, however, that the fit function does not apply for very small preparation rates; at zero preparation rate, the preparation would never end. It is also not wise to extrapolate the fit function too far beyond the data on the right; generating a negative effectiveness is a clear indication of going too far.

The following variations to the data were performed and had only minor effects on the results (the slope changed by up to 10 percent, and the goodness-of-fit parameters changed slightly):

- Restriction of inspection defects to major defects [4] rather than all defects found.
- Removing any one of the data points.
- Use of three separate selection criteria on the test defects:
  - Total test defects.
  - Code-only test defects, including generated code.
  - Code-only test defects, excluding generated code.

In all of these cases, the effectiveness is well fit by a straight line that achieves 100 percent at zero preparation rate and declines by a constant amount for every 100 lines of code per labor hour of inspection preparation review (see Figure 1 on page 9).

Cost Model

The cost model should include all major costs that are affected by the inspection preparation rate. At the top level, this cost model includes the costs associated with just two development processes:

1. Code inspections with inspection rework.
2. Testing with test rework and regression testing.

A more detailed analysis, including other code development processes or including the effect of code complexity on cost could not be supported due to absence of necessary data in the program databases. Code complexity, both algorithmic complexity and interface complexity, would be expected to affect inspection effectiveness as well as inspection rework labor hours and test rework labor hours. These were excluded from this model due to the absence of appropriate supporting data.

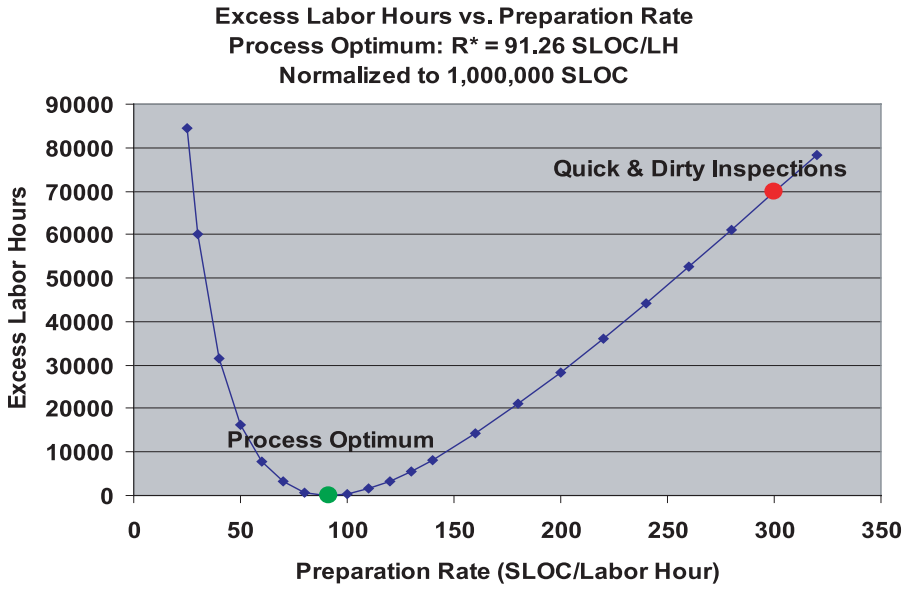


Figure 2: *Murphy's Tongue - Excess Labor as a Function of Inspection Preparation Rate*

In this model, total cost is the sum of the following items:

- Code inspection preparation labor.
- Code inspection meeting labor.
- Code inspection rework labor.
- The labor for running the full test suite once.
- Testing rework labor.
- Regression testing labor.

In deriving the cost function, the following statements are assumed to be approximately true:

- The inspection preparation labor is proportional to the amount of code being reviewed.
- The inspection meeting labor and the inspection rework labor are proportional to the number of major defects found in the inspections.
- The test rework labor and the regression test labor are proportional to the number of major defects found during testing.
- A significant number of the major code defects found by testing could have been found earlier in code inspections.

Using these assumptions, the total excess cost due to discovered defects can be expressed as follows:

$$T_e = C + S*W_{ri}*I + S*D/R + S*I* (-m)*(T_r*E_t - W_{ri})*R$$

See the Cost Function Derivation sidebar for a more detailed derivation of this cost function and for variable and terms definitions.

Cost Optimization

The optimum code inspection preparation rate is obtained by finding the point at which the cost function has a minimum. A minimum is found by setting the first derivative of the total cost with respect to code inspection preparation rate equal to zero, solving for code inspection preparation rate, and verifying that the point is a minimum and not a maximum or an inflection point:

$$I*S*(-m)*(T_r*E_t - W_{ri}) - D*S/R^2 = 0$$

The result is the following formula for the optimum preparation rate:

$$R^* = SQRT\{D/[I*(-m)*(T_r*E_t - W_{ri})]\}$$

For demonstration purposes, the following values are used:

$D = 4$

$E_t = 1.0$ approximately (a very high quality requirement: man rated or species rated)

$I = 0.040$ defects/SLOC

$m = -0.00075$ (SLOC/labor hour)$^{-1}$

$S = 1{,}000{,}000$ SLOC

$T_r = 20$ labor hours/defect

$W_{ri} = 4$ labor hours/defect

Substituting this demonstration data yields the following optimum code inspection preparation rate:

$$R^* = 91.29 \text{ SLOC/labor hour}$$

The relative cost in excess of the cost at the optimum rate is given by evaluating the difference in the total cost at a given rate minus the cost at the optimum rate:

$$T_M = S*D*(1/R - 1/R^*) + S*I*(-m)*(T_r*E_t - W_{ri})*(R - R^*)$$

Since this curve is shaped like a parabola that opens upward (see Figure 2), there must be a unique preparation rate R^* that minimizes the total cost. Figure 2 is named in memory of Murphy's Law because any variation from the optimum, no matter how well intentioned, will increase development costs.

The Cost of Variation

It is not enough to know the optimum preparation rate. No human process is without variation, so it is necessary to know the cost of variation. Whether a code inspection is run slightly too fast or slightly too slow, the cost is higher than the minimum cost. This idea is summarized in the following definition: "World class quality is on target with minimum variance." [5]

The cost of variation is approximately parabolic with respect to deviations from the optimum preparation rate. Therefore the Taguchi cost-of-variance formula for this model is the averaged second derivative term in a Taylor expansion of the cost function [6],

$$T_v = (D*S/R^{*3})*\langle(R - R^*)^2\rangle$$

where the angle brackets indicate computing the average over the whole dataset. Substituting example data yields the following:

$$T_v = 5.26*\langle(R - R^*)^2\rangle \text{ labor hours}$$

This is an approximation to the exact cost behavior for this model, but it shows the salient point that variation itself has a cost that may be worth minimizing.

Conclusions

If a program collects the right data from the inspection, test, and cost-accounting processes, performance and cost analysis can result in the ability to predict program cost and quality performance in terms of one inspection process control, inspection preparation rate. With the right data, a second driver could be added to the model, e.g., code complexity. This driver would be expected to affect inspec-

Cost Function Derivation

Code inspection labor consists of preparation labor, inspection meeting labor, and inspection defect rework labor. Preparation labor is just size divided by preparation rate: S/R. The inspection meeting labor is linearly related to the preparation labor because the meeting time will be driven by the number of candidate defects to be discussed and recorded. The sum of preparation labor and meeting labor is C + D*S/R, where C and D are the linear regression coefficients. Inspection rework is driven by the number of defects found, and that is the discoverable defect insertion rate "I" times the amount of code inspected "S" times the inspection effectiveness "(1 + m*R)" times the labor to fix an average defect "W_{ri}."

Testing labor includes the cost of testing perfect code "T_o*S" (running through the whole test suite once), rework driven labor, and regression testing. Both regression testing and test rework will be driven by the number of defects detected during testing: the number escaping the inspections "I*S*(-m)*R" times the test effectiveness "E_t" times the labor for fixing and regression testing an average defect "T_r." Symbolically this can be expressed as follows:

$$T_c = T_o*S + C + D*S/R + W_{ri}*I*S*(1 + m*R) + T_r*E_t*I*S*(-m)*R$$

or

$$T_e = T_c - T_o*S = C + S*W_{ri}*I + S*D/R + S*I*(-m)*(T_r*E_t - W_{ri})*R$$

where:

C = Y-intercept of empirical fit of total inspection labor (preparation plus meeting).

D = Slope of empirical fit of total inspection labor vs. preparation rate.

Et = Test effectiveness (defects found in test/defects found in test plus those found after test completion excluding all defects that cannot be found by testing, e.g., requirements defects).

I = Discoverable code defect rate = code inspection defect rate + test defect rate.

m = Slope of code inspection effectiveness regression line.

R = Code inspection preparation rate (SLOC/labor hour).

S = Total SLOC inspected.

SLOC = Non-comment, non-blank, physical lines of code. Any consistently used size measure will work, e.g., executable statements, function points, etc.

T_o = Labor for testing one line of perfect code (regression testing not needed).

T_c = Total cost (labor hours).

T_e = Total excess cost due to discovered defects.

T_r = Labor per defect to do test rework and regression testing.

W_{ri} = Labor hours to rework a code inspection defect.

C+D*S/R = Labor for inspection preparation plus the inspection metting.

I*S = Defects present at code inspection.

(1+m*R) = Code inspection effectiveness.

I*S*(-m)*R = Defects missed by the code inspection that escape into test.

I*S(1+m*R) = Defects found by the code inspection.

S/R = Inspection preparation labor.

T_0*S = Total labor for testing perfect code (registration testing not needed).

$T_r*E_t*I*S*(-m)*R$ = Labor to do test rework.

$W_{ri}*I*S*(1+m*R)$ = Labor for reworking defects found during the inspection.

tion effectiveness as well as inspection rework labor hours and test rework labor hours.

Realistically, one ought to be able to achieve 90 percent code inspection effectiveness by preparing at a cost optimizing rate of about 100 SLOC/labor hour without any other change in the code inspection process. To achieve further improvement, say 99 percent effectiveness without increasing code inspection cost, it would be necessary to improve the code inspection process. There are several ways to do this:

- Use checklists that are improved each time they are used.
- Train each developer to develop and to use an individualized checklist.
- Use Watts Humphrey's Team Software Process/Personal Software Process that does both of the above and more [7].

In the example using the optimal 91.29 SLOC/labor hour preparation rate rather than a *quick and dirty* 300 SLOC/labor hour, there is a savings of nearly 70,000 labor hours/MSLOC ($ millions/MSLOC at any realistic labor rate). If the code inspection process could be modified to 99 percent efficiency without increasing code inspection costs, then the savings would potentially exceed 107,000 labor hours/MSLOC. Such potential improvement makes clear the value in performing code inspections at a deliberate pace and reliably recording certain key information to enable optimization of program performance. With this kind of analysis and with reliably recorded data, simultaneously working better, faster, and cheaper really is possible!

In conclusion, to get the best value from an inspection, it is more cost effective to prepare for that inspection as if preparing for a final exam in college rather than reading the material as if reading a light novel for entertainment. However, preparation thoroughness consistent with a Ph.D. thesis defense may not be appropriate unless there is a clear business case for exceptional quality, e.g., man-rated or better.◆

Acknowledgments

The author would like to thank John Gibson of Lockheed Martin Mission Systems and Pat Dorazio, Earl Pape, and Bernie Pindell of Lockheed Martin Management & Data Systems, Gaithersburg, Md., for reading this report and making numerous helpful suggestions. Thanks are also due to Dr. Abol Ardalan of The University of Maryland University College for teaching me the value of cost models and how they are built, and to Dr. Gary Kaskowitz of The University of Maryland University College for teaching me basic business statistics.

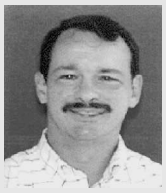
References

1. Fagan, Michael G., Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems Journal*, vol. 15, no. 3, 1976.
2. Fagan, Michael G, Advances in Software Inspections, *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, July 1986.
3. Montgomery, Douglas C. and Runger, George C., *Applied Statistics and Probability for Engineers*, John Wiley & Sons, Inc., NY, 1994, pp. 315-317, 493-495, and 510-513. Also see the Microsoft Excel-97, SR-2 online help for the LINEST function, example 4, using the F and R2 Statistics.
4. See note 1.
5. Wheeler, Donald J. and Chambers, David S., *Understanding Statistical Process Control*, 2nd Ed., SPC Press, Knoxville, TN, 1992, pp. 141-147.
6. Ibid., pp. 143-147.
7. Humphrey, Watts S.; Lovelace, Mark; and Hoppes, Ryan, *Introduction to the Team Software Process*, Addison-Wesley Publishing Co., 1999, ISBN: 020147719X.

Notes

1. The Lockheed Martin Management & Data Systems definition of a major defect is a defect that will cause a malfunction or deviation from requirements or specifications, seriously violates policies or standards, indicates missing function, or makes the system unusable. The defect must be fixed since it may cause some degree of project failure, economic loss, poor customer satisfaction, or contractual or legal breach.
2. A computer software configuration item is a large set of related functionality produced by one team of developers.

About the Author



Bob McCann is a staff systems engineer at Lockheed Martin Management & Data Systems in Gaithersburg, Md. He has nearly 20 years of experience in computational physics and high performance computing, including nine years at Princeton Plasma Physics Laboratory working in the U.S. DOE-controlled fusion program, as well as about 10 years experience in design and development of relational databases of various kinds. McCann is currently a member of the M&DS Metrics Process Steering Committee and works on improving software development processes, methods, and metrics. He has a bachelor's degree in physics with a concentration in mathematics from Shippensburg University; master's degrees in physics and computer science from University of Maryland and Southwest Texas State University, respectively; and is working on a master's degree in computer systems management/software development management at the Univeristy of Maryland University College.

Lockheed Martin Management and Data Systems
700 North Frederick Avenue
Gaithersburg, MD 20879
Phone: (301) 240-4273
Fax: (301) 240-7190
E-mail: bob.mccann@lmco.com

"It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free."

—Steve McConnell

"The most important single aspect of software development is to be clear about what you are trying to build."

—Bjarne Stroustroup

Software Estimating Model Calibration

Dr. Randall Jensen
Software Engineering, Inc.

The last decade has brought increasing pressure on software model developers to include a calibration capability in their models that will reduce errors and improve software development estimates. An invalid underlying assumption is that software estimate errors are primarily due to weaknesses in the estimating technology (models). Data errors and the impact of estimator capability are never considered to be significant error sources. The intent of this article is to raise awareness of software estimate errors sources, and to objectively consider the real impacts of model calibration.

A common phrase used in many software presentations during the past few years is, "We need properly calibrated and validated models." This is an assumption that a perfect estimating model will eliminate the risk in scheduling and managing software projects. Unfortunately, the calibrated, validated model is only the tip of the estimating iceberg.

There are four error classes that exist in any software estimate: estimating technology (model), environment, project data, and the estimator. The estimating technology is centered on the estimating model or tool. No model is perfect; however, several widely used models are viable estimating tools.

The second error class involves the environment where the software is to be developed. The environment includes the development system such as tools and practices, operating system, physical facilities, organization structure, and management. This information, which is generally sketchy when the project acquisition estimates are made, is detailed and firm at the time of contract award.

Project data, the third error class, is a large error source in all estimates. Size is a fundamental estimating tool parameter and is usually specified in source lines of code. Effective size, which defines the project magnitude, is so difficult to establish for almost all estimates that alternate estimate forms such as function points and objects have become popular size predictors. No size predicting method is trivial, and none of the methods have proven accuracy advantages.

The last of the four error classes is the estimator. The estimator collects the estimate data, establishes the estimating model input parameters, performs the cost and schedule analysis, and produces the final estimate. A realistic estimate requires that the estimator be proficient with the estimating tool. Proficiency equates to training and experience. Training requires more than access to a User's Guide, and experience is not instantaneous.

The estimator and the project data are the largest error contributors. Technology is actually the least significant source of error. Any of the major models or tools in the hands of an experienced estimator can produce realistic software development estimates. The estimating tool is quite analogous to a scalpel in the hands of a surgeon.

Calibrated, Validated Tools

Those responsible for producing estimates typically make four simple assumptions. First, the estimator is not an accuracy issue. Second, the project data used to develop or calibrate the model is of high quality. Third, the environment is constant and not an issue ("Despite this cost variation, COCOMO does not include a factor for management quality, but instead provides estimates which assume the project will be well managed." [1]). Finally, the estimating technology (model) is assumed to be the major error source.

The phrase "properly calibrated and validated models" does not appear to be the elixir that eliminates the errors and risk in scheduling and managing software projects. Calibrated, validated models are still necessary for realistic software estimates, just as sharp scalpels are necessary in good surgery.

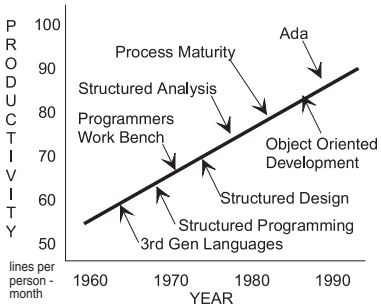
The estimating model must fit the data from which it is defined. This data must represent the estimating model application area. Developing a model from commercial data processing projects and applying the resulting model to spacecraft control should not be expected to produce acceptable cost and schedule predictions.

The major models are capable of estimating a wide range of software projects. These tools, in general, have been calibrated (and validated) with considerable project data over a long period of time. Internal environment adjustment factors account for variations between products and environments. For example, the number of environment adjustments in Sage and SEER-SEM is near 30. It is important to note these factors have also been validated. External model calibration changes the meanings of these factors and negates the model validation, as we will see later.

One argument against using traditional estimating tools is that traditional models were all developed in 1970s and 1980s. Fortunately, the software development industry has not changed significantly since then. True, we have much improved software development approaches and environments, but have these environments improved development productivity?

Defense industry software productivity, measured from the start of development through final qualification test, has grown almost linearly from 1960 through the present. A simplified (smoothed) productivity growth curve in Figure 1 shows this growth. The result, smoothed or not,

Figure 1: *Average Software Development Productivity Growth from 1960 to 1990*



shows a growth of software development productivity less than one source line per person-month per year during the entire 30-year period. For that period of time, each new technology has assured us the productivity problems of the past have been solved.

The traditional models, in the hands of experienced estimators still produce accurate and high-quality cost and schedule projections.

The Calibration Issues

The major calibration issue is what should be calibrated – the estimating model, the development organization, or the estimator. The industry trend leans toward the technology solution: the estimating model. The problem with the technology solution is calibration changes the model. This change is easily demonstrated with one of the simpler models, the Constructive Cost Model (COCOMO).

COCOMO vs. REVIC

The COCOMO software-estimating model was first published in 1981 [2]. The development effort *E,* defined by the embedded software version of the model, is given by Equation 1,

$$E = 2.8\prod_{i=1}^{16} f_i S_e^{1.2} \qquad (1)$$

where the impact of the product and the development environment is specified by the product of 16 effort adjustment factors

$$(\prod_{i=1}^{16} f_i)$$

and the effective development size . A variation of the COCOMO model, known as REVIC [3] was introduced a few years later. REVIC was developed from a collection of U.S. Air Force project data and defined development effort by the relationship shown in Equation 2,

$$E = 3.312\prod_{i=1}^{16} f_i S_e^{1.2} \qquad (2)$$

that differs from COCOMO (Equation 1) only in the proportionality constant. The REVIC equation is equivalent to

$$E = 2.8(k_1)\prod_{i=1}^{16} f_i S_e^{1.2}$$

where the *calibration factor* k_1 =1.18. Note the REVIC definitions of the 16 environment adjustment factors (EAFs) are identical to the COCOMO definitions. REVIC was validated as a new estimating model using USAF project data. It is apparent that REVIC is a calibrated COCOMO model, and the calibration required that REVIC be revalidated before use to provide confidence in REVIC's cost and schedule estimates. The philosophical question, "Can any estimating model be changed and used without revalidation?" is a major concern. Many software-estimating tools contain calibration constants. Can these constants be changed from their default value without requiring model revalidation?

One serious model impact introduced by calibration is the redefinition of the model itself. We can illustrate the effect by factoring the analyst capability rating (ACAP) out of the Effort Adjustment Factor (EAF) product to obtain the equivalent effort relationship shown in Equation 3,

$$E = 2.8(k_1)(ACAP)\prod_{i=1}^{15} f_i S_e^{1.2} = 2.8(1.0)(1.0)\prod_{i=1}^{15} f_i S_e^{1.2} \qquad (3)$$

For example, consider a software developer who uses COCOMO to produce realistic software estimates. The default calibration constant for COCOMO is 1.0. A manager notices the ACAP assumes the development organization to be average (*ACAP* = 1.0), which is unacceptable from a business standpoint.

The organization, according to the manager, should be considered to be in the 90th percentile category (*ACAP* = 0.71). Since the effort projections were realistic before the change to the ACAP, the calibration factor must be increased to 1.41 to rebalance the equation. The resulting effort equation is given by,

$$E = 2.8(1.41)(.71)\prod_{i=1}^{15} f_i S_e^{1.2} = 3.95\prod_{i=1}^{16} f_i S_e^{1.2}$$

Changing the proportionality constant from 2.8 to 3.95 is not the only change to the estimating model. The analyst capability definition was changed to allow average analysts to be rated in the upper 10 percentile as well. Is the model defined by the new effort equation equivalent to COCOMO? The models cannot be considered equivalent because the EAF definitions and the model definitions are different. Should the new equation define a new model that requires validation? Yes.

Calibration Database

The calibration (or validation) project database is also a major calibration issue. The database must be of sufficient size to effectively validate the new or modified estimating model. Inadequate data leads to inadequate validation. Yet, the calibration constant used in many models allows the validation database to be as small as a single project.

The data in the project database must also represent a common data definition; that is, defined with the same data rule set. For example, the task size definition must be consistent across all the development tasks in the database. It would be irrational to define part of the projects in terms of total size, part as modified size, and part as effective size.

The development environment for each task must be defined in the project database since no two tasks are developed under identical conditions. Some tasks may be developed in the Ada programming language, some may have severe timing constraints, and some may have a volatile requirements set. This information is necessary to adjust the environment factors in the selected model before a valid estimate can be produced or before the model can be validated (calibrated).

The U.S. Air Force Space and Missile Center, with assistance from the Space Systems Cost Analysis Group, developed a software project database that contains more than 2,800 contributed and unverified data records. The software project database is the largest and most widely used source of software project data. Some of the individual records provide fairly complete descriptions of each development task. This database has been the primary source of data for model calibration. Unfortunately, much of the data is incomplete, the definitions used by the individual data points are inconsistent, and the information necessary to extract effective size is not included. Thus, only a small portion of the software project database data is useful.

Estimator Impact

The most important variation in software estimates is the training, skill, and experience of the estimator. The impact of the estimator on an estimate is almost always

ignored. Many estimators are not trained in estimating or in using their tools, which are assumed to be user-friendly (most are); the estimator need only to set values for the model parameters. The estimates are accepted without sanity checks to ascertain the estimate realism. It is easy to set model parameters, but sometimes quite difficult to set those parameters correctly. A slightly positive bias on all 16 COCOMO parameters can significantly change the resulting cost and schedule estimate.

Experience as an estimator or a software developer also has a great impact on the parameter values inserted into the estimate. Is it possible to understand development system volatility without some knowledge of the software development process? What is the impact of reuse or COTS on the effective size of the development? These questions demand experience and skill.

Decalogue Project

The Decalogue Project is the current stage of an ongoing software model calibration project that was started and first published in the early 1990s [4]. The project results have been published from 1995 through 1999 as Air Force Institute of Technology master's theses, and in other publications such as CROSSTALK [5].

The project applies statistical methods to determine the accuracy of several software models. The current project results are not encouraging; however, the study results should be expected because of the underlying assumptions. The four simple estimate assumptions regarding estimators, data, environment, and estimating technology are all implicit in the Decalogue Project. The first indication that the Decalogue Project had problems was that *none* of the estimating models calibrated in the study had reasonable accuracy.

Before condemning the estimating model performance, the assumptions and experimental methods of the Decalogue Project must be understood. The most critical parameter in an estimate is the task size. The models use an effective size that is a weighted combination of new source code, original (existing) code, and modifications to the original code. The size data precision is very important to eliminate size inaccuracy from the estimate error.

The size data extracted from the software project database for the Decalogue Project was the total size data, not the effective size data that is used by the models. As an example of the estimate error introduced by using total size data, consider a system upgrade of 1,000 source lines in an existing 100,000-line software system. The development effort is more closely related to the 1,000-line upgrade than to the 100,000 existing lines. Development productivity is a simple test of the size data; that is, the ratio of size to development effort. More than half of the tested data points in the software project database yielded a productivity of more than 1,000 lines per person month. This productivity is an order of magnitude greater than that achieved in a typical project today.

The development environment was poorly specified or missing from the software project database. To compensate for the poor environment data, each model was set to a nominal environment description for the estimates. The nominal environment was essentially the model default.

Each software model was assigned to a graduate student for the analysis. Student quality is not an issue. However, the students had little, if any, specific model training or estimating experience.

The poor results achieved by the Decalogue Project demonstrate that poor calibration (validation) data, coupled with misused models and inexperienced estimators, lead to invalid estimates.

Conclusion

There are some fallacies related to the calibration of software estimating models. The first fallacy is that calibration data is generally accurate, consistent, and unbiased. Good validation data is carefully verified to ascertain its completeness, conformance to the database definitions, and accuracy. Size data in the calibration database must be complete enough to allow effective size to be extracted from the new, original, and modified size components.

Continued on page 18

Members of the German Air Force enjoyed the Opening Welcome Reception.

STC 2001 provided an excellent networking opportunity at all levels.

Winners of the first CrossTalk Airplane Contest (see sidebar on page 30).

Beatles enthusiasts young and old enjoyed 1964's live tribute.

Photos C
Randy S
On-Line
Software
Suppor
a
MSgt Neil
Public Affa

Attendees heard one of 199 software-related presentations.

The crowd watched eagerly at the first CrossTalk Airplane Contest.

NOLOGY CONFERENCE 2001
RIL - 3 MAY

Courtesy of
Schreifels
Services
Technology
rt Center
and
Werenskjolt
irs Specialist

Continued from page 15

Effective size, as used in the models, is necessary to predict cost and schedule. Organizational experience with the software also impacts effective size. Since organizations are not equal and unchanging, environment data is also necessary to perform estimates. Good validation data is not readily available.

The second fallacy is that estimates are independent of estimator training and experience. That is not likely. One cannot assume estimates produced by real people are ever above the significant biases introduced by lack of training, skill, or experience.

The third fallacy is that properly calibrated and validated tools eliminate major estimate errors; wrong again. Estimating tools are simply estimating tools. Proper validation reduces errors inherent in the tool itself. The model, or tool, is like a good scalpel. The quality of the estimate lies primarily in the quality of the user.

On the other hand, what disadvantages does calibration provide?

- Calibration invalidates the model. Model calibration requires revalidation before the model can be used with confidence.
- Calibration can destroy the meaning of the model's environment parameters. Unless that is the calibration goal, care must be taken to ensure the parameter definitions are intact.
- Calibration cloaks weakness in the estimator. Errors introduced by improperly using a model can be compensated for through the calibration process.
- Calibration makes it impossible for estimators using different versions (calibrations) of a model to compare results. This problem can get very severe when correlating estimates from the acquisition team and the contractor.

The bottom line is another phrase used in many software presentations during the past few years: "We need properly calibrated and validated models." Yes, we certainly need validated tools, but we also need trained, skilled, and experienced estimators.◆

References

1. Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, Inc., 1981, pg. 487.
2. Boehm, B. B., *Software Engineering Economics*, Prentice-Hall, Inc., 1981.
3. Kile, R.L., *REVIC Software Cost Estimating Model Users Manual,* Ver 9.0, February 9, 1991.
4. Ourada, G. L. and Ferens, D. V., Software Cost Estimating Models: A Calibration, Evaluation, and Comparison, *Cost Estimating and Analysis: Balancing Technology and Declining Budgets*, New York, Springer Verlag, 1992, pp. 83-101.
5. D. V. Ferens and Christensen, D. S., Does Calibration Improve Predictive Accuracy? *CrossTalk*, April 2000, pp. 14-17.

About the Author

Randall W. Jensen, Ph.D., is president of Software Engineering, Inc., and specializes in software project resource management. Dr. Jensen developed the model that underlies the Sage and the GAI SEER-SEM software cost and schedule estimating systems. He received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. Dr. Jensen has published several textbooks, including *Software Engineering,* and numerous software and hardware analysis papers. He has bachelor's, master's and doctorate's degrees in electrical engineering from Utah State University.

Software Engineering, Inc.
660 North Highland Blvd.
Brigham City, UT 84302
Phone: (435) 734-2585
Fax: (435) 734-2586
E-mail: seisage@aol.com
www.seisage.com

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

—Maurice Wilkes

A Smart Way to Begin a Civilian Engineering Career in the U.S. Air Force

Tracy Stauder
Software Technology Support Center

The U.S. Air Force has an entry-level professional development program for civilian scientists and engineers called PALACE Acquire (PAQ). The program was established to heighten the Air Force's ability to maintain the leading edge in today's technology-intensive environment by hiring dynamic, creative, and innovative scientists and engineers. This article describes the PAQ program for science and engineer professionals, the qualification requirements for the program, and the benefits to Air Force organizations that participate in the program.

In today's information technology age, good engineers are at a premium. Companies nationwide are facing the challenge of attracting and retaining qualified engineering professionals. The U.S. government's challenge is even greater as they must compete with industry. When recruiting college graduates, one *carrot* or recruiting advantage that the Air Force has over industry is its PALACE Acquire program for scientists and engineers.

I recently had the opportunity to become trained as a PAQ recruiter and thought CROSSTALK would be a great vehicle to get the word out to others in the Air Force who may also be facing challenges in recruiting entry-level engineers. Hence, this article is intended to help others in the Air Force understand the PAQ program better and the variety of benefits that it offers the employee as well as the hiring organization. In addition, many undergraduate engineering students through their college coursework are exposed to CROSSTALK. Hopefully, this article will also reach those who have not yet made a career employment decision.

PAQ Program for Scientists and Engineers

Science and engineering professionals provide a broad foundation of expertise required to develop and support the Air Force's technological needs. To sustain the national defense effort on the leading edge of explosive technological advancement, the Scientist and Engineer Professional Development Program provides an extensive two- or three-year training program.

PAQ program participants with a bachelor of science (BS) degree are placed in a three-year training program. During the first and third years, the participant works full-time obtaining practical on-the-job experience. During the second year he or she pursues full-time graduate study. Participants with a completed master of science (MS) degree or a BS degree supplemented by one year of professional engineering or science experience, enter a two-year work experience training track.

A broad range of science and engineering disciplines support the PAQ program. These include the following:

- Aerospace Engineering
- Computer Science
- Electrical Engineering
- Electronics Engineering
- Mechanical Engineering
- Operations Research

Individuals with a completed BS degree enter the trainee program at the GS-07 level. A grade point average of 3.05 and a GRE score of 1000 are required to ensure acceptance into graduate school. Individuals with a completed MS degree, or a BS degree with one year's experience, enter at a GS-09. Successful completion of either the two- or three-year track results in a promotion to the GS-12 journeyman level. Career promotions are based on satisfactory completion of specific training criteria as outlined in a formal training plan. For additional information on government employment grade structures and salaries, see the Office of Personnel Management web site at www.opm.gov.

What Sets PAQ Apart?

Three-year PAQ participants have their graduate salary, tuition, and books paid for by the Air Force. They also earn their salary at the same time they are a full-time student. What a great deal! The participants' jobs are to go to school. They must take job-related courses, fulfill MS degree requirements, and be in school for one academic year. If unable to finish a degree program in one year, they are encouraged to finish on their own time.

By attending graduate school, the participant incurs an obligation for continued employment as they will be required to sign an agreement to continue in federal service for three times the length of the total academic training period.

Participants also have access to a full-range of recreational and support facilities open to Air Force employees such as social clubs, hobby shop, fitness centers/gyms, golf course, credit union, library, and family support center to name a few.

PAQ Package: A Quick Look

Qualification Requirements Summary

Participants in the Air Force's PALACE Acquire program require the following:

- U.S. citizenship.
- Geographic mobility.
- Overall undergraduate GPA = 3.05 or greater.
- GRE verbal and quantitative score =1000 or greater.
- Degree from Accreditation Board for Engineering and Technology, Inc. (ABET) approved institutions.

Employee Benefits

PAQ participants are full-time permanent civil service employees receiving the same benefits and entitlements as most permanent full-time federal employees. These include the following:

- 10 paid holidays.
- 13 days vacation (0-3 years of service); 20 days (3-15 years); 26 days (15+years).
- Sick leave, up to 13 days annually.
- Performance based bonuses and time-off awards.
- Health and Life Insurance options.
- Federal retirement, Social Security, and Medicare.
- Thrift Savings Plan (civil service 401(k) plan).

Organization Benefits

The organization that trains a PAQ employee benefits as well. In particular, the employee's time and training for the two- or three-year internship is paid for by the Air Force's central salary account. In other words, there is no cost to the local organization that is responsible for training and developing the participant. Another big benefit is that the participants are highly motivated, innovative, and dynamic scientists and engineers.

What Participants Say

Patrick Warren, who earned his master's degree from Ohio State University through the program in March, and who is also an engineer in the Aeronautical System Center Engineering Directorate, says that the benefits of the PAQ program were just too good to pass up.

Anthony Spohn, who is on course to attend Ohio State in the fall to pursue a graduate degree in Aerospace Engineering, has not seen anything like this program before. He says, "Other than the government, I've never seen a program like this anywhere. Boeing, Lockheed, yeah, they'll reimburse you after you go to class, but they aren't going to pay your salary while you're gone. They aren't going to pay for your education and books while you are going to class. I don't have to spend any money out of pocket, and I get a salary. You don't get this anywhere else that I've seen."

The opportunities for education and training were especially important to Spohn. "It was the ultimate driver in my job seeking."

Conclusion

The U.S. Air Force employs more than 15,000 civilian scientists and engineers working in laboratories, test centers, system development offices, and depots. Many engineers directly support pilots, crewmembers, aircraft, spacecraft, and weapons systems. As a civilian engineer in the Air Force, I have found the opportunities in my workplace to be just as challenging and rewarding as those I had when working for a defense contractor. I am proud to be associated with the PAQ program and hope that other Air Force managers will see the value in this program as well as engineering students soon to make a career decision.◆

Additional Information

For more information or for an application to the PAQ program, please contact the PAQ administrator at Headquarters Air Force Personnel Center, Civilian Career Management Directorate at Randolph Air Force Base, Texas.

HQ AFPC/DPKCW
Attn: PAQ Administrator
555 E Street West, Suite 1
Randolph AFB, TX 78150-4530
Commercial: 210-565-2252
Toll Free: 1-800-847-0108, Ext. 3025
E-mail: secp@afpc.randolph.af.mil
Web Site:
www.afpc.randolph.af.mil/cp/secp/palacq.htm

Reference

1. Air Force Instruction-AFI 36-602 Personnel Civilian Intern Programs, July 25, 1994.

About the Author

Tracy Stauder is a technical program manager for the Air Force's Software Technology Support Center (STSC) at Hill AFB. She supports the STSC in its efforts of publishing CROSSTALK, hosting the annual Software Technology Conference, and offering hands-on consulting. She has a master's and a bachelor's degree in electrical engineering, from Southern Illinois University. She has worked for the Software Engineering Division at Hill AFB in Utah for the past eight years. She has also worked as an avionics engineer at McDonnell Douglas in St. Louis for eight years.

Software Technology Support Center
00-ALC/TISE
7278 4th Street
Hill AFB, UT 84056-5205
Phone: (801) 775-5746
E-mail: tracy.stauder@hill.af.mil

Web Sites

Software Technology Support Center

www.stsc.hill.af.mil

The STSC is the command focus for proactive application of software technology in weapon, command and control, intelligence and mission-critical systems. It helps organizations identify, evaluate, and adopt technologies that improve software product quality, production efficiency, and predictability.

Test and Measurement World

www.tmworld.com

This is the on-line version of Test & Measurement World and Test & Measurement Europe, which cover the electronics testing industry, providing how-to information for engineers who test, measure, and inspect electronic devices, components, and systems.

Software Engineering Institute

www.sei.cmu.edu

The SEI is a federally funded research and-development center sponsored by the U.S. Department of Defense to provide leadership in advancing the state of the practice of software engineering to improve software quality. The SEI staff has extensive technical and managerial experience from government, industry, and academia.

Institute of Electrical and Electronics Engineers

www.ieee.org

The IEEE promotes the engineering process of creating, developing, integrating, sharing, and applying knowledge about electrical and information technologies and sciences. IEEE provides technical publications, conferences, career development assistance, financial services and more.

American Society for Quality

www.asq.org

The ASQ is dedicated to the ongoing, development, advancement, and promotion of quality concepts, principles, and techniques. ASQ has more than 120,000 individual and 1,100 organizational members.

Process Capability Data for the Asking

Lt. Col. Robert Lang
Defense Contract Management Agency

To gauge a contractor's process maturity (on individual programs), the Defense Contract Management Agency has applied the Software Engineering Institute's Capability Maturity Model®. While being incrementally deployed this effort is already paying benefits to program offices, contractors, and the Department of Defense. The goal remains continuous process improvement to ensure the war fighter, the end user, receives the highest quality systems.

Wouldn't it make sense to have a way for government program offices to determine the maturity of a contractor's software development process without incurring the cost and time to conduct a total software capability evaluation? Wouldn't it be efficient to have a way to eliminate redundant reviews of contractor software development processes by different government offices?

Well, now there is a way to obtain this data. Just simply ask. While not fully operational until next year, the capability to provide such data is currently in place at almost half of the field locations within the Defense Contract Management Agency (DCMA).

As the on-site government representatives at contractor facilities, DCMA provides assistance to all branches of the military. Its scope of effort is defined within the Federal Acquisition Regulation (FAR). A complete description of DCMA capabilities was previously described in CROSSTALK [1]. They include the evaluation and surveillance of contractor management systems such as the processes used in software development [2]. For this, the agency has adopted use of the Software Engineering Institute's (SEI) Capability Maturity Model® for Software (SW-CMM).

The SW-CMM is the language we needed to speak, and speak fluently, to communicate with the broad range of customers across the Department of Defense (DoD). It is the language spoken by government program offices when conducting software capability evaluations for source selections or lesser reviews. It is the language selected by the DoD to reduce risk on acquisitions [3]. It is the language employed by contractors when conducting a CMM-based appraisal for internal process improvement (CBA IPI).

® The Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office.

CMM-Based Insight

Our initiative to speak this common language, what we call CMM-based insight, is simple in concept. Taking advantage of DCMA's in-plant presence, we will primarily organize daily observations into findings based on the CMM. Observations undergo an internal peer review for conformity to the CMM, then data is freely shared with the applicable contractor and passed to program offices. Findings will be used to concentrate DCMA effort based on risk. Details concerning the process, responsibilities, and outcomes are captured in the Method Description Document (MDD), which is available on line at www.dcma.mil/onebook/4.0/4.3/initiatives.htm.

The goals (see Table 1) directly benefit program offices, contractors, and the DoD. Regardless of DCMA location, program offices will have consistent data concerning a contractor's software process maturity for programs within DCMA cognizance. Since data is freely shared with the contractor, concern or disagreement on high-risk areas can be resolved at the working level, or elevated as necessary to the DCMA/Contractor/Program Office Management Council [4]. The data can be used in future process reviews to reduce or eliminate redundant areas. The results from this continuous review could also be used as a vehicle to ensure that contractors have maintained a process capability level per DoD policy [5] or in support of independent expert program reviews of software intensive systems [6].

CMM Based Insight Goals
1. Provide program and software development process risk information to DCMA and buying commands
2. Promote supplier process improvements based on trend analysis of CMM based observations
3. Consistently maintain data to identify process capability in support of source selection and contract monitoring
4. Promote DCMA internal process improvements

Table 1: *CMM-Based Insight Goals*

Evaluation Relationships

CMM-based insight is not a software capability evaluation or a CBA IPI (see Table 2). While data could be used to substantiate another evaluation, DCMA will never rate a particular company through CMM-based insight. The initiative is focused on identifying areas of concern on individual programs (i.e., higher risk process areas) and allocating the appropriate level of resources commensurate with that risk.

Incremental Phases

As previously discussed, the initiative is simple in concept. But like the process of teaching an adult to speak (and think) in a new language, making this transition has

Table 2: *Comparison of Evaluation*

	Software Capability Evaluation	CMM - Based Appraisal for Internal Process Improvement	DCMA CMM - Based Insight
Basis of Evaluation	Software CMM	Software CMM	Software CMM
Company Rating Provided	No	Yes	No
Frequency	One time	One Time	Continuous
Data Refreshed	No	No	Yes (18 Month Max)
Conducting Organization	Typically government (including DCMA)	Contractor	Government
Basis of review	Sponsor selected, usually within a particular domain	Representative programs across a business base	All programs within DCMA cognizance

involved a culture change in DCMA software surveillance activities. As such, incremental phases (see Table 3) were designed to assist the transition.

Phase I validated the approach at the home locations of our agency Software Engineering Institute (SEI) affiliates. Phase II verified that approach for suitability and effectiveness in a typical field environment. Phase III will verify the capture and transmission of data before the initiative is implemented agency wide.

Data Organization Callenges

The primary purpose of Phase II was to verify the approach. Due to the shear number of inputs – necessary for the correlation of observations to the applicable key practices, internal peer reviews, identification, and subsequent action on high risk areas – the need for an adequate support tool was recognized early. (This situation will be resolved in Phase III when data collection is incorporated into the common tool supporting the entire DCMA Risk Assessment Management Program.) Despite this burdensome data collection, currently 45 percent of our field locations have volunteered as pilot locations and converted operations. Why? It is because of the benefits realized. These are perhaps best illustrated with actual examples.

Example 1 - Improvement Not Rating: A program office concerned with a history of poor software quality wanted the contractor to operate at CMM Level 3. The contracting company's upper management believed the company was well within these parameters and retained an outside consultant to verify this position. Initial results indicated the contractor was operating at CMM Level 3. The DCMA field office disagreed, however, based upon observations and findings per the CMM.

Working with the program office, the findings were questioned and the issue elevated to upper management. The program office held that if the review revealed a significantly different result than that observed in day-to-day operations, the government would sponsor an independent software capability evaluation. If the government evaluation revealed the contractor was more interested in paper ratings than software quality improvement, the government would consider developing a second source for the procurement.

What was the end result? The final evaluation revealed operations at CMM Level 1. The contractor was well on the way to Level 2, but far from the desired Level 3 target profile. Was this a typical contractor/government confrontation? Quite the opposite, it fostered a spirit of process improvement. For the first time, there was an accurate and understood baseline. The contractor developed a roadmap for process maturity and during the course of two years, achieved the desired Level 3 profile. DCMA, the government on-site representative, participated in the mini reviews and was a team member on the final contractor-conducted CBA IPI.

Example 2 - Risk Based Operations: If a correlation between capability (maturity level) and actual performance (cost, schedule, and technical) [7] is accepted, it would seem reasonable to assume that there is less government surveillance of higher maturity operations than those with lower maturity. In the absence of data, however, people often focus on those areas where they are comfortable. Consequently, a low-risk area might get as much attention as a high-risk area.

This is not so with the CMM-based insight methodology because it is based on data and focuses expended effort in proportion to risk. This is the case at one of our pilot locations where the contractor has achieved CMM Level 5. The CMM-based insight data will be used to ensure that DCMA effort and resources are allocated to the areas of highest risk.

Example 3 - The Rest of the Story: Is a CMM rating truly representative of all programs at a given facility? As the name states, the model measures a capability. It would seem logical to assume that if a capability has been demonstrated on one program, that it has been applied to all. With mandated levels, though, there are other pressures that come into play. At one pilot location the contractor had conducted a CBA IPI that resulted in a finding of CMM Level 3. The contractor had selected programs across the business base and then hung a banner over the building entrance saying "CMM Level 3 Certified." What was wrong with that?

First, the rating Level 3 certified is confusing and misleading. Certified by whom? Secondly, the review did not include the largest program, one which had been experiencing problems at the international level. While the CBA IPI shows a company's capability to operate at a given level, it is not necessarily true for all programs. It should be, and seems to be in most cases, especially when the focus is on process improvement. However in this particular case, it was not. With the DCMA data, the banner was removed and the applicable program office understood that operations on their program were not at CMM Level 3, and why.

Example 4 - Eliminate/Reduce Duplicative Reviews: Concerned about software quality, a joint program office planned a review of the contractor's software development processes. The DCMA pilot location, a front-runner for this initiative, already had the data in the com-

Table 3: *CMM Based Insight Implementation*

Start	Phase	Task	Product	Who	New Sites	Total Sites
Oct 99	I	-Develop & Validate Approach	-Method Description Document (MDD) -Training Material	DCMA SEI Affiliates	4	4
Jun 00	IIA	-Verify Procedures -Validate Data Collection -Learn & employ methodology	-Updated MDD -Finalize Data Requirements	Volunteer Field Locations	5	9
May 01	IIB	-Refine approach -Learn & employ methodology	-Procedure Update (as required) -Trained personnel	Volunteer Field Locations	10	19
Aug 01	IIC	-Refine approach -Learn & employ methodology	-Procedure Update (as required) -Trained personnel	Volunteer Field Locations	5-10	24-29
TBD (Estimated Winter 01/02)	III	Verify Data Integration into DCMA Risk Assessment Management Program	-Data collection tool supporting CMM based operations	All Pilot Sites	0	24-29
TBD (Estimated Spring 02	IV	Agency Wide Deployment	Process Data in terms of SW-CMM	All DCMA locations	13-18	42

mon language of the CMM. It clearly identified strengths and weaknesses. The review was cancelled and the DCMA data was used in follow-on actions with the contractor. This is only one example, but the dollar savings across the department quickly add up. The cost to conduct a software capability evaluation has been estimated at $50,000 for both the government and contractor [8].

Experience and Training

A complete process is defined as having 1) procedures and methods for defining the relationship of tasks, 2) tools and equipment, and 3) people with skills, training, and motivation [9]. The first two elements have already been addressed. Concerning people, the agency has more than 400 personnel supporting software quality assurance. To assure this workforce is properly prepared to deliver consistently first-rate assessments, we have instituted a multiphase development program:

- Basic Training: The agency's formal training is called the DCMA Software Professional Development Program. Individuals proceed through two training levels. Level one requires completing 72 hours of computer-based training, 40 hours of classroom instruction, and a formal mentoring program focused on practical application of course material. Level 2 requires an additional 97 hours of computer-based training, 120 hours of classroom instruction, and further mentoring. The SEI's CMM is integrated into the computer-based training, classroom instruction, and mentoring. Currently 78 percent of agency software personnel have obtained Level 2 status. To maintain this level, individuals must complete a minimum of 12 hours of software-related training each year.
- Application Training: As each field location begins operating under the CMM-based insight initiative, all personnel undergo an additional 20 hours of specific application training conducted on site by the DCMA Software Center. Applicable contractors and government program offices have been welcomed into this training. It is specifically focused on the initiative's implementation and daily operation. The material was developed by the DCMA Software Center and is currently being reviewed by the SEI.
- On Call Assistance: DCMA personnel have direct access to the six-person DCMA Software Center. In addition, one-eighth of the total field workforce has completed the SEI's Software Capability Evaluation training. Additional assistance is available to any of our evaluators from highly qualified agency personnel who are SEI-certified lead assessors.
- Implementation Measurement: Training provides a foundation for conducting business per the CMM, but it does not directly correlate to experience, which can only come with time. Progress in implementing this initiative has been promising. For instance, more and more of our personnel have been requested as team members by companies when conducting CBA IPIs. However, to gauge implementation progress for this initiative across the entire agency and to make necessary adjustments, the agency is employing a top-level metric based on key process area coverage. Progress will be reviewed by the agency director, his senior leadership team, and DCMA field commanders.

CMM-Based Insight and CMMI

The baseline for our efforts is the SW-CMM. We fully expect, and are making preparations, to switch over to the Capability Maturity Model Integrated℠ (CMMI℠) at a later date. The agency is part of the SEI-led CMMI Steering Group responsible for developing the SW-CMM/CMMI turnover within the DoD [10]. For CMM-based insight, the transition should incur little breakage moving to the integrated model. The biggest challenge in using either model is the discipline and knowledge of application – both of which we are gaining with our current effort and are fully transferable. Field sites coming aboard in each phase are shown in Table 4.

DCMA Creditability

A past issue of CROSSTALK raised the point, "A Level 3 development effort coupled with a Level 1 acquiring effort often equates to a Level 1 delivery capability; yet the Level 3 developer is often blamed, and the Software (SW) CMM is cited as inadequate[11]." I saw this firsthand, with disasterous results, as a junior officer. So how does DCMA measure up?

To answer that question, we took the sister capability maturity model, the Software Acquisition CMM, and tailored it for DCMA use. We pilot tested and made adjustments as applicable. We then went agency wide, conducting reviews from November 1999 until April 2000. Eight equally qualified teams were used to maintain consistency. What were the results? There were a few organizations operating at the defined level but predominately the field offices within the agency operate at the performed level (Level 1).

Table 4: *Pilot Locations*

DCMA Pilot Locations

Phase I (Beginning Oct 99)

- Boston, Nashua
- Delaware Valley, PA
- Denver
- Syracuse

Phase II (Added Jun 00)

- Boeing, Philadelphia
- Lockheed Martin Owego, N.Y.
- Lockheed Martin Sunnyvale, CA
- St Petersburg
- Sikorsky

Phase II B (May 01)

- Birmingham, Huntsville
- Bell Helicopter, Textron
- Northrup Grumman, Bethpage
- Northrup Grumman, Melbourne
- San Antonio, NASA, Houston
- Hartford
- Baltimore, Mannassas
- Boeing, St Louis
- Orlando, Harris, Melbourne
- Springfield, N.J.

Phase II C (Summer 01) – Up to 10 Volunteer Locations

More importantly, we established a solid baseline and each location has a detailed roadmap for improvement per the model structure. Field locations have been working improvements and the first round of follow-on appraisals began in the spring of 2001. The original evaluation team members constitute the personnel pool to support independent evaluation of improvements similar to the industry approach with a CBA IPI.

Conclusion

DCMA was always required and continues to conduct evaluations of contractor's software development processes per the FAR. The agency is now deploying a standard methodology via continuous process evaluations that is organized in the CMM, the common DoD language, and is based on the day-to-day observations of the in-plant DCMA personnel. Findings are peer-reviewed, and all data is freely shared with the applicable contractor and is available to government program offices.

While full agency implementation will not occur until spring 2002, the approach has been developed with SEI affiliates and is currently in pilot testing at 45 percent of DCMA field locations. Program offices, the contractors, and the DoD are already realizing benefits. So, how much does the agency believe in using this approach to gauge contractor operations? Enough so that we are walking the walk and measuring our operations to the same framework.◆

References

1. Holt, Kevin E., Software Acquisition Support in the Defense Contract Management Command, CROSSTALK, March 1997.
2. Federal Acquisition Regulation, Part 42, Section 302(a)(41).
3. USD(AT&L) Memorandum, Software Evaluations for ACAT I Programs, Oct. 26, 1999.
4. Malishenko, Timothy, Management Councils Emerge as Valuable Asset in the Program Manager's Tool Kit, *Program Manager Magazine*, March/April 1999.
5. USD(AT&L) Memorandum, Software Evaluations for ACAT I Programs, Oct. 26, 1999.
6. USD (AT&L) Memorandum, Independent Expert Program Reviews of Software Intensive System Acquisition, Dec. 21, 2000.
7. Lawlis, Dr. Patricia K.; Flowe, Capt. Robert M.; Thordahl, Capt. James B.; A Correlation Study of the CMM and Software Development Performance, CROSSTALK, September 1995.
8. SCE Reuse: Ending Redundant Reviews, *Acquisition Reform Today*, Vol 3, No.1, January/February 1998.
9. Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley Publishing Company, 1994, pg. 9.
10. Deputy Under Secretary of Defense (S&T) letter dated Dec. 11, 2000, Use of CMMI Evaluations by the Department of Defense.
11. Jarzombek, Lt. Col. Joe, Integrating Acquisition with Software and Systems Engineering, CROSSTALK, August 1999.

About the Author

Lt. Col. Robert Lang currently serves as the director of the Defense Contract Management Agency Software Center. He has 20 years of active duty service in the U.S. Air Force in various acquisition specialties. His previous assignment was program manager for the Iceland Air Defense System. He has a bachelor's degree in engineering technology from Norwich University, a master's degree in engineering management from Western New England College, and is a graduate of the Defense Systems Management College, Advanced Program Management Course.

DCMAC-G
495 Summer Street
Boston, MA 02210-2184
Phone: (617) 753-3739
E-mail: rlang@dcmde.dcma.mil

Coming Events

July 7-13
2nd Int'l Symposium on Image and Signal Processing and Analysis ISPA'01
ispa.zesoi.fer.hr

July 22-26
JAWS S³ Symposium
www.jawsswg.hill.af.mil

August 1-5
0HCI International 2001: International Conference on Human-Computer Interaction. 1st International Conference on Universal Access in Human-Computer Interaction
hcii2001.engr.wisc.edu

Aug 27-30
Software Test Automation Conference
www.sqe.com/testautomation

August 27-31
Fifth IEEE International Symposium on Requirements Engineering
www.re01.org

September 10-14
Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering
www.esec.ocg.at

February 4-6, 2002
International Conference on COTS-Based Software Systems (ICCBSS) At the Heart of the Revolution
http://www.iccbss.org

April 28 - May 3, 2002
STC 2002
"Forging the Future of Defense Through Technology"
www.stc-online.org

Getting Software Engineering into Our Guts

Lawrence Bernstein and David Klappholz
Stevens Institute of Technology

Many if not most, computer science students are enamored of technology (state of the art), but averse to the discipline of software process (state of the practice). Staying up late hacking code and eating pizza is great fun for them, while following the discipline of software engineering best practice is decidedly not. We have developed a methodology, Live-Thru Case Histories, for overcoming this aversion, and have found it to be very productive in a pilot study. We are developing the methodology further for use both at universities and in industry.

Software projects often fail because too many software folks think that software engineering *process* is just bureaucracy. Often software people lack software engineering education, or when they had it, they fought it. To compound the problem, they do not accept software best practices. Our challenge is to overcome the natural biases of software professionals.

We tried lecturing on case histories of failed software projects, but these lectures and associated readings only convince many students of others' stupidity. They do not internalize the lessons. Others intellectually accept the existence of the problems, but just reading about them does not convert many at the gut level where one sits up, takes notice, and does things differently.

At the gut level, successful software project managers instinctively anticipate problems and take steps to avoid them. The challenge is to educate software people so that they do not have to hone their instincts through the "school of hard knocks."

Our approach is to force students to live through specific case histories, each one chosen to get across a small number of important issues. This method works. Students internalize the software engineering lessons and follow best practices in their next projects to avoid the traps they experienced.

Here is how the approach works. First, select a set of software process issues. These are the ones we chose for our first live-through case history:

- The need to have close customer/user relations.
- The need for up-to-date documentation throughout the life of the project.
- The need to identify risks and to develop contingency plans.
- The need to account for human foibles.

Second, choose a case history based on a project facing these challenges. Do not give students the entire case history up front; rather, give them the same problem as the actual developers who executed the case history faced. Give the students no more information about the problem than the original developers had at the start. You may simplify information to ease understanding.

Background

Computer science is the study of the technology (state-of-the-art) involved in the development of computer software. As it is usually taught, computer science deals with *programming in the small*, i.e., one-person or few-person software projects. Software engineering, on the other hand, is the study of the method or process (state-of-the-practice) whereby production software is developed – *programming in the large*. State-of-the-practice includes both engineering practices and project management or group dynamic processes.

Typical computer science programs offer a software engineering or senior project course as a capstone. Due to the very different natures of technology vs. method/process, and because computer science students are typically technology-oriented and process-averse, the typical software engineering course reaches far fewer future software developers than suits the best interests of either the students or the software industry. Thus we developed a novel instructional method, the Live-Thru Case History method for addressing this problem. We have developed a first live-through case history and have used it successfully in the first few weeks of a two-semester undergraduate software-engineering course.

The result was that students were shocked into an awareness of the issues and how to deal with them in only six weeks of twice-weekly class meetings. One class meeting each week was devoted to individual unstructured project meetings, and the other to lectures on software engineering topics, including other case histories.

Conducting the Case History

There would be just one live-through case history in our senior project course, so we had to choose one that would achieve the greatest effect in the limited time available. We chose the case history of a brief development project that one of the authors worked on in 1985 as a public service project. The project was automating an elementary school library's manual system for generating overdue-book notices.

The class of 40 students was divided randomly into four equal-size development teams. Students were given the same details possessed by the original software developers in the case history. The instructor role-played the customer, the school librarian, and was available to respond to students' questions, both in class and by e-mail. Students were told that the customer would evaluate their work exactly as it would be evaluated in the real world.

Results

As is frequently the case in real software development projects, the overdue book notice project had a hidden requirement. That requirement was so obvious to the customer that she failed to mention it; overdue notices must be sorted first by teacher name, then for each teacher by class, and finally, within each class by student's family name. The system analyst rejected the real software system when she first saw it. The original developers failed to elicit the hidden make-or-break requirement, and thus failed to satisfy it. Each of the student teams fell into this same trap thus learning the lesson of the need to find any hidden requirements.

Students also learned of the need for high-quality documentation and contingency planning due to the real-world phenomenon of attrition through illness, death, relocation, etc. At the project midpoint, students were rotated. A student from each team judged by the instructor to be the team's strongest developer and another chosen randomly were removed from the team and reassigned to a different team.

To evaluate each team's success in adapting to the simulated attrition, students were asked to describe what they would have done differently after the case study project was complete. About 75 percent of the students mentioned the importance of up-to-date documentation. Nearly 20 percent had developed insight into appropriate staff utilization, including the use of "understudies" and preparing for incorporating new team members. They demonstrated the learned value of these processes.

An evaluation of how well the students internalized the need for solid requirements engineering was performed at the end of the live-through case history. Students completed a written exam based on another case history that included a more difficult requirements engineering problem than that of the overdue book notice project. About 75 percent of the students demonstrated they had mastered the notion of hidden requirements, and about 33 percent showed they had achieved reasonable competence in requirements engineering; about 10 percent showed extremely keen insight into the problem.

The innovative process of live-through case histories is more effective than the traditionally taught software engineering course. In it, students were given lectures, homework, and exams based on a well-respected software engineering text. Then they were asked to develop a project. However when they approached the project, they could not readily apply the learned techniques. Once they understood the need for the processes, they relearned them as they tried to apply them.◆

Directions

The authors request that those teaching software engineering use the Live-Thru Case Histories in their courses and report on the results. These materials are available at www.njcse.org/Projects/Live_Thru_Case_Histories/Materials_For_Live_Thru_Case_Histories.htm, along with a complete paper describing the live through approach in detail.

Please participate in gathering data to support or refute the claims in this paper. It is our intent to use the experience of instructors in several venues to make anecdotal conclusions more meaningful and perhaps statistically significant. We invite those who agree with us to join a consortium for the purpose of creating additional case histories and helping to refine the process.

The Great Learning Process (Confucius)

Things being investigated (piloted), knowledge became complete.

Their knowledge being complete, their process was updated.

Their process being updated, their practices were cultivated.

Their practices being cultivated, their projects were regulated.

Their projects being regulated, their organizations were rightly governed.

Their organizations being rightly governed, the whole corporation was made tranquil and prosperous.

Adapted slightly by and with apologies from Tim Powell

About the Authors

David Klappholz has 27 years of experience teaching computer science and performing and supervising technology research sponsored by such organizations as National Science Foundation, Department of Energy, IBM Research, and The New Jersey Commission on Science and Technology. He has been on the computer science faculties of Columbia University and Polytechnic University, and is currently on the computer science faculty at Stevens Institute of Technology, in Hoboken, N.J., and associate director of the New Jersey Center for Software Engineering.

Department of Computer Science
Stevens Institute of Technology
Castle Point Station
Hoboken, NJ 07030
Phone: (908) 464-0805
E-mail: d.klappholz@worldnet.att.net

Lawrence Bernstein is a former vice president of AT&T where he managed small-, medium-, and large-scale software projects, both commercial and military, for 35 years. He is a Fellow of both the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery. He is currently senior industry professor of Software Engineering at Stevens Institute of Technology, in Hoboken, N.J., and director of the New Jersey Center for Software Engineering.

Department of Computer Science
Stevens Institute of Technology
Castle Point Station
Hoboken, NJ 07030
Phone: (973)258-9213
E-mail: lbernstein@worldnet.att.net

"Before software can be reusable it first has to be usable."
— Ralph Johnson

The Problem with Testing

Norman Hines
JE Sverdrup Naval Systems Group

Testing is inefficient for the detection and removal of requirements and design defects. As a result, lessons learned in testing can only help prevent defects in the development of subsequent software and subsequent process improvement. Instead of testing out defects to achieve quality measures, quality should be designed into software. Thus test development should parallel the development of the software it tests.

Unlike other engineering disciplines, software development produces products of undetermined quality. Testing is then used to find defects to be corrected. Instead of testing to produce a quality product, software engineers should design in quality [1]. The purpose of testing should not be to identify defects inserted in earlier phases, but to demonstrate, validate, and certify the absence of defects.

Beginning with the Industrial Revolution, many technical fields evolved into engineering fields, but sometimes not until after considerable damage and loss of life. In each case, the less scientific, less systematic, and less mathematically rigorous approaches resulted in designs of inefficient safety, reliability, efficiency, or cost. Furthermore, while other engineering practices characteristically attempt to consciously prevent mistakes, software engineering seems only to correct defects after testing has uncovered them [2].

Many software professionals have espoused the opinion that there are "always defects in software [3]." Yet in the context of electrical, mechanical, or civil engineering the world has come to expect defect-free circuit boards, appliances, vehicles, machines, buildings, bridges, etc.

Follow the Basics

All models of the software development life cycle center upon four phases: requirements analysis, design, implementation, and testing. The waterfall model requires each phase to act on the entire project. Other models use the same phases, but for intermediate releases or individual software components.

Software components should not be designed until their requirements have been identified and analyzed. Software components should not be implemented until they have been designed. Even if a software component contains experimental features for a prototype, or contains only some of the final system's features as an increment, that prototype or incremental software component should be designed before it is implemented.

Software components cannot be tested until after they have been implemented. Defects in software cannot be removed until they have been identified. Defects are often injected during requirements analysis or design, but testing cannot detect them until after implementation. Testing is therefore inefficient for the detection of requirements and design defects, and thus inefficient for their removal.

Testing in the Life Cycle

Burnstein, et al. have developed a Testing Maturity Model (TMM) [4] similar to the Capability Maturity Model® [5]. The TMM states that to view testing as the fourth phase of software development is at best Level 2. However, it is physically impossible to test a software component until it has been implemented.

The solution to this difference of viewpoint can be found in TMM Level 3, which states that one should analyze test requirements at the same time as analyzing software requirements, design tests at the same time as designing software, and write tests at the same time as implementing code. Thus, test development parallels software development. Nevertheless, the tests themselves can only identify defects after the fact.

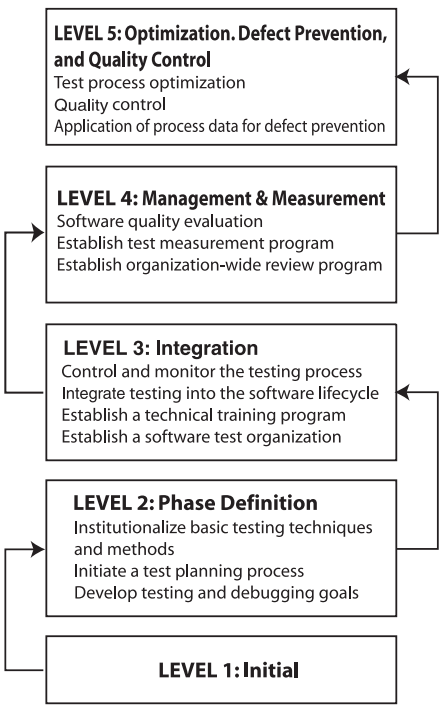
Furthermore, testing can only prove the existence of defects, not their absence. If testing finds few or no defects, it is either because there are no defects, or because the testing is not adequate. If testing finds too many defects, it may be the product's fault, or the testing procedures themselves.

Branch coverage testing cannot exercise all paths under all states with all possible data. Regression testing can only exercise portions of the software, essentially sampling usage in the search for defects.

The clean-room methodology uses statistical quality certification and testing based on anticipated usage. Usage probability distributions are developed to determine the systems most likely used most often [6]. However, clean-room testing is predicated upon mathematical proof of each software product; testing is supposed to confirm quality, not locate defects. This scenario-based method of simulation and statistically driven testing has been reported as 30 times better than classical coverage testing [7].

Page-Jones dismisses mathematical proofs of correctness because they must be based on assumptions [8], yet both testing and correctness verification are done against the software's requirements. Both are therefore based on the same assumptions; incorrect assumptions result in incorrect conclusions. This indictment of proofs of correctness must also condemn testing for the same reason.

Table 1: *TMM Levels*



The clean-room methodology's rigorous correctness verification approaches zero defects prior to any execution [9], and therefore prior to any testing. Correctness verification by mathematical proof seems better than testing to answer the question, "Does the software product meet requirements?"

Properly done test requirements analysis, design, and implementation that parallels the same phases of software development may help in early defect detection. However, done improperly (as when developers test their own software), this practice may result in tests that only test the parts that work, and in software that passes its tests but nevertheless contains defects. Increasingly frustrated users insist that there are serious defects in the software, while increasingly adversarial developers insist that the tests reveal no defects.

Test requirements analysis done separately from software requirements analysis can make successful testing impossible. A multi-million dollar project was only given high-level requirements, from which the software developers derived their own set of (often-undocumented) lower-level requirements, to which they designed and implemented the software. After the software had been implemented, a test manager derived his own set of lower-level requirements, one of which had not even been considered by the developers. The design and test requirements were mutually exclusive in this area, so it was impossible for the software to pass testing. This failure scrapped the entire project and destroyed several careers [10].

Defect Removal and Prevention

Test-result-driven defect removal is detective work; the maintenance programmer must identify and track down the cause within the software. Defect removal is also similar to archeology, since all previous versions of the software, and documentation of all previous phases of the development may have to be researched, if available. Using testing to validate that software is not defective [9], rather than to identify and remove defects, moves their removal from detection to comparative analysis [7].

TMM Level 3 integrates testing into the software lifecycle. This includes testing each procedure or module as soon as possible after each is written. Integration testing is also done as soon as possible after the components are integrated. Nevertheless, the concept of defect prevention is not addressed until TMM Level 4, and then only as a philosophy for the entire testing process.

Testing cannot prevent the occurrence of defects; it can only aid in the prevention of their recurrence in future components. This is why neither CMM nor TMM discusses actual defect prevention, or more accurately, *subsequent* defect prevention until Level 5. Waiting until one has reached Level 5 before trying to prevent defects can be very costly, both in terms of correcting defects not prevented and in lost business and goodwill from providing defective software.

Waiting until implementation to test a component for defects caused in much earlier phases seems too much of a delay; yet, an emphasis in testing for defect prevention is exactly that. An ounce of prevention may be worth a pound of cure, but one cannot use a cure as if it were a preventative.

There are several methods currently available to accomplish defect prevention at earlier levels of maturity such as Cleanroom Software Engineering [9], Zero Defect Software [11], and other provably correct software methods [2, 12].

Software Quality and Process Improvement

Gene Krinz, Mission Operations' director for the NASA space shuttle, is quoted as saying about the quality of the flight software, "You can't test quality into the software [11]." Clean-room methods teach that one can neither *test in* nor *debug in* quality [9].

If quality was not present in the requirements analysis, design, or implementation, testing cannot put it there. One of TMM's Level 3 maturity goals is software quality evaluation. While many quality attributes may be measured by testing, and many quality goals may be linked to testing's ultimate objectives, most aspects of software quality come from the quality of its design.

Procedure coupling and cohesion [13], measures of object-oriented design quality such as encapsulation, connascence, encumbrance, class cohesion, type conformance, closed behavior [8], and other quantitative measures of software quality, are established in the design phase. They should be measured soon after each component is designed; do not wait until after implementation to measure them with testing.

Some authors have suggested that analyzing, designing, and implementing tests in parallel with the products to be tested will somehow improve the processes used to develop those products. [3] However, since the software product testers should be different from those who developed it, there needs to be some way for the testers to communicate their process improvement lessons learned to the developers. Testers and developers should communicate effectively; every developer should also act as a tester (but only for components developed by others).

Designing in Quality

One of the maturity subgoals of subsequent defect prevention is establishing a causal analysis mechanism to identify the root causes of defects. Already there is evidence that most defects are created in the requirements analysis and design phases [11]. Some have put the percentage of defects caused in these two phases at 70 percent [3].

Clear communication and careful documentation are required to prevent injecting defects during the requirements analysis phase. Requirements are characteristically inconsistent, incomplete, ambiguous, nonspecific, duplicate, and inconstant. Interface descriptions, prototypes, use cases, models, contracts, pre- and post-conditions, etc. are all useful tools.

To prevent injecting defects during the design phase, software components must never be designed until a large part of their requirements have been identified and analyzed. The design should be thorough, using such things as entities and relationships, data and control flow, state transitions, algorithms, etc. Peer reviews, correctness proofs and verifications, etc. are good ways to demonstrate that a design satisfies its requirements.

Preventing the injection of defects during the implementation phase requires that software components never be implemented until they have been designed. It is far too easy to implement a software component while the design is still evolving, sometimes just in the developer's mind. Poor documentation and a lack of structure in the code usually accompany an increased number of defects per 100 lines of code [3]. As I mentioned earlier, this applies even to prototype and incremental software components; those experimental or partial features should be designed before implementation.

The clean-room method has an excellent track record of near defect-free software development, as documented by the Software Technology Support Center, Hill AFB, Utah, regardless of Daich's statements to the contrary [3]. Clean-room is compatible with CMM Levels 2 through 5 [9], and can be implemented in phases at all these levels [6].

Conclusion

It is my dream that software engineering will become as much of an engineering discipline as the others; users will have just as much confidence that their software is as defect free as their cars, highway bridges, and aircraft.

Testing should be used to demonstrate the absence of defects, not to identify defects inserted in earlier phases. It should be used to certify that the software components implement their designs, and that these designs satisfy their requirements.

Analyzing testing requirements should be done in parallel with analyzing the software components' requirements. Tests should be designed in parallel with designing the components. Test implementation should occur in parallel with implementing the components, and developing integration tests should be done in parallel with integration.

The source of software defects is a lack of discipline in proper requirements analysis, design, and implementation processes. Testing must physically occur after implementation, so reliance on it to detect defects delays their correction. Until software defects are attacked at their source, software will continue to be developed as if it were an art form rather than a craft, engineering discipline, or a science.◆

References

1. Humphrey, W. S., Making Software Manageable, *CROSSTALK,* December 1996, pp. 3-6.
2. Baber, R. L., *The Spine of Software: Designing Provably Correct Software: Theory and Practice,* John Wiley & Sons Ltd., Chichester, United Kingdom, 1987.
3. Daich, G. T., Emphasizing Software Test Process Improvement, *CROSSTALK,* June 1996, pp. 20-26, and Daich, Gregory T., Letters to the Editor, *CROSSTALK,* September 1996, pp. 2-3, 30.
4. Burnstein, I.; Suwannasart, T.; and Carlson, C.R., Developing a Testing Maturity Model: Part I, *CROSSTALK,* August 1996, pp. 21-24; Part II, *CROSSTALK*, September 1996, pp. 19-26.
5. Paulk, M. C.; Curtis, B.; Chrissis, M. B.; and Weber, C. V., *Capability Maturity Model℠ for Software, Version 1.1,* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1993.
6. Hausler, P. A.; Linger, R. C.; and Trammel, Adopting Cleanroom Software Engineering with a Phased Approach, *IBM Systems Journal,* volume 33, number 1, 1994, p. 95.
7. Bernstein, L.; Burke Jr., E. H.; and Bauer, W. F., Simulation- and Modeling-Driven Software Development, *CROSSTALK,* July 1996, pp. 25-27.
8. Page-Jones, M., *What Every Programmer Should Know About Object-Oriented Design,* Dorset House Publishing, New York, New York, 1995.
9. Linger, R.C., Cleanroom Software Engineering: Management Overview, *Cleanroom Pamphlet,* Software Technology Support Center, Hill Air Force Base, Utah, April 1995.
10. Unpublished CMM Tutorial, information withheld to protect the people involved.
11. Schulmeyer, G. G., *Zero Defect Software,* McGraw-Hill, Inc., New York, New York, 1990.
12. Martin, J., *System Design from Provably Correct Constructs,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.
13. Page-Jones, M., *The Practical Guide to Structured Systems Design,* second edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

About the Author

Norman Hines is a software developer for JE Sverdrup, working at the Naval Air Warfare Center, China Lake, Calif. He is currently working on projects that integrate weapon simulation systems with actual range data in real time and post mission. He has more than 20 years experience in software development, as well as a bachelor's degree in mathematics and business administration from University of Wisconsin–Platteville, a master's degree in business administration from University of Michigan, and a master's degree in computer science from California State University, Chico.

JE Sverdrup Naval Systems Group
900 N. Heritage Dr.
Ridgecrest, CA 93555
Phone: (760) 939-9460
E-mail: nwsverdrup@navair.navy.mil

"Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

— Donald Knuth

Paper Airplane Contest Wows STC 2001 Conference Attendees

SALT LAKE CITY – The audience surrounding the foyer at the Salt Lake Convention Center erupted into a giant "ahhhhh," when the paper airplane sailing effortlessly from the platform launch landed 73 and one-half feet down the makeshift runway below them. Although he didn't know it yet, Steve Smith had just won first place at the first ever CROSSTALK Paper Airplane Contest held May 1 as part of the 13th Annual Software Technology Conference (STC).

For his winning flight, Smith, engineer project manager at the Joint Force Program Office, took home a Handspring Visor Prism handheld computer. In second place, with 70 and three-fourths feet, was John Stark, chief engineer, Predictive Technologies Division, Science Applications International Corp., who received a $100 gift certificate to Barnes & Noble. Third place went to Maj. Stewart Laing, deputy chief of Testing Division, Global Transportation Network Program Management Office, with a flight of 65 and two-thirds feet. Laing won a $50 gift certificate to the Olive Garden.

In its first year, the paper airplane contest drew 105 entrants who constructed their airplanes from a choice of two paper weights pre-printed and supplied by the Software Technology Support Center (STSC) at Hill Air Force Base and the contest sponsor SHIM Enterprise, Inc. The STSC held the airplane contest as a gathering event to solicit authors for its monthly journal CROSSTALK, *The Journal of Defense Software Engineering*. Contest participants were cheered on by about 350 spectators eating pizza, kabobs, and desserts while listening to running contest commentary by co-announcers Dave Cook and Les Dupaix, engineering consultants at the STSC.

Nearly 3,000 people attended the STC conference, "2001 Software Odyssey: Controlling Cost, Schedule and Quality," offering 199 software-related presentations along with 167 trade show exhibits. The conference is co-sponsored by the Departments of the Air Force, Army, Navy, Defense Information Systems Agency and Utah State University Extension.◆

The Weakest Geek

I made an interesting observation at the Software Technology Conference last month. Members of the digerati congregate to determine who among them is the Alpha Geek – the person with the most technological prowess.

Bring together software engineers for a technical discussion at a conference and watch the circumlocution fly. After respectful introductions, potential Alpha Geeks expatiate opinions like Cliff Clavin articulating the mating habits of the Yellow-Bellied-Sap-Sucker to the gang at Cheers.

You have to be shrewd, primed, and agile to run with the big geeks. Hesitate with a banal response, mention an archaic technology, or offer a feeble theorem and your status will drop faster than a California power line.

One gaffe and you are demoted to a nebbish Beta Geek. To stay in the clique, Beta Geeks offer blandishments for Alpha Geek ideas like Gilligan in a conversation with the other Castaways on how to get off the island, e.g., "that's a good idea Professor … Skipper has a good idea … I like Mary Ann's idea."

Once these cliques start there are three ways to disband them. The first is free food. Food trumps technological prowess on a geek's hierarchy of needs. So all dialogue ceases while participants amass large amounts of caffeine and sugar – the breakfast of Alpha Geeks. This, however, is a provisional break. Once geeks restock, there is a short period of indolence, then the logorrhea resumes.

The second solution can only be employed once during the conference. This occurs when exhibits are opened. All discussion is curtailed and a free-for-all occurs for the coolest vendor giveaways. Only rock concerts, holiday shopping, and downloading music before the Napster court decision, rival the furry of this frenzy.

Since most conference attendees have no control over food or exhibits, their only option to break up a clique is to mention configuration management, documentation, or testing. Drop one of those topics into the discussion and potential Alpha Geeks will chortle and make a graceful exit. Beta Geeks experience temporary paralysis as they search for a complimentary comment. However, do this too many times and you will reach Cipher Geek status – the geek no one wants to talk to.

Lets face facts here, no matter how valuable configuration management, documentation, and testing are to a project, no one likes to talk about them. Like taking out the trash, washing dishes, and cleaning bathrooms – no one wants to do it, but it has to be done. Anyone thinking otherwise has malodorous software.

Take configuration management. It's obvious to me that designing software is more invigorating than making sure versions are in order. Yet anyone knows that hotshot designers could not design their way out of a paper bag without a good configuration management system. To design otherwise is analogous to spinning plates on the Ed Sullivan show. At first it looks cool, then the plates start to wobble, and it gets real ugly as plates shatter on the floor. So configuration management can be cool as long as someone else is doing it.

How about documentation? Now there is a paradox. Why can't creative, knowledgeable, bright designers clarify their designs in their native language? Maybe they got into the software business so they wouldn't have to deal with English.

Testing may be an engineer's biggest blight. Although a very noble and necessary part of the profession no other assignment impairs growth, withers ambitions, or impedes one's progress and prosperity more than testing. Once you have ventured into testing you become branded as a second-class engineer. I'm not saying its right or fair, but it happens. Colleagues ostracize you as a miscreant as you wonder the halls in a company issued testing smock – you know how hard it is to clean software stains.

It would be easy to wallow in pity but my observation is that most test engineers are not helping their cause. Their pusillanimous approach to the mockery reinforces the image. I have one message for test engineers – wake up!

It's the 21st century and you need to realize you have all the cards. You have the final say. You can expose hotshot designer's inaccuracies. You have the power! Designers should truckle at your feet in hopes of obtaining favor upon their designs.

I know this is hard to conceptualize after decades of scoffing. Therefore I'm offering you an exemplar. Her name is Anne Robinson, the host of the insufferable game show "The Weakest Link." Take note of her portentous attitude as she fires verbal laser jabs at her subjects.

Discard the days of meekly handing in problem reports and cowering to prima designers. It is time to hold court with those pompous colleagues. Line them up. Have them defend their designs in excruciating detail. Expose their flaws. Make them sweat.

You can start your own test cliques. Swap stories of grilling designers, compare kickbacks, and discuss the latest smock styles. You know you have arrived when a design engineer joins the group to talk shop and you taunt, "You are the weakest geek. Goodbye."◆

— Gary Petersen, Shim Enterprise, Inc.

CrossTalk / TISE
5851 F Ave.
Bldg. 849, Rm B04
Hill AFB, UT 84056-5713

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