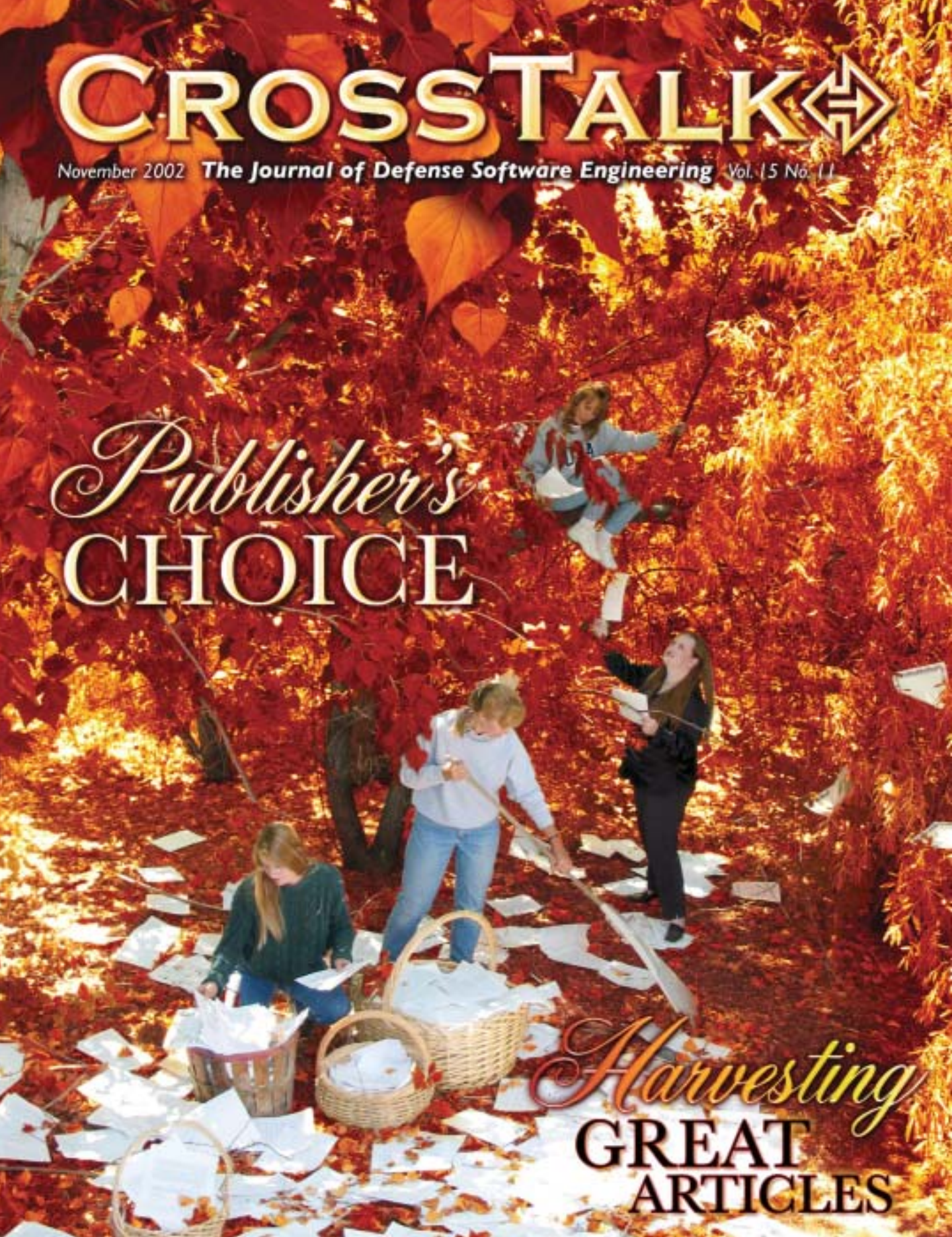

CROSSTALK
November 2002 The Journal of Defense Software Engineering Vol. 15 No. 11
Publisher's CHOICE
Harvesting GREAT ARTICLES

Best Practices

On The Cover
Cover Design by Kent Bingham.

Software Engineering Technology

Open Forum

Online Articles

Departments

CrossTalk

Sponsor	*Lt. Col. Glenn A. Palmer*
Publisher	*Tracy Stauder*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Bowers*
Associate Editor	*Chelene Fortier*
Article Coordinator	*Nicole Kentta*
Creative Services Coordinator	*Janna Kay Jensen*
Phone	(801) 586-0095
Fax	(801) 777-8069
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk
CRSIP Online	www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 25.

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

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

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

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

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

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

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

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

Well, Actually, Associate Publisher's Choice

I especially enjoyed putting together this month's issue. Each month I am the one responsible for deciding which articles will be published in CrossTalk. I lobbied for this month's theme, "Publisher's Choice," so that I would have an opportunity to share several very good articles that have been waiting to be published for far too long. The only problem is that there still was not enough space to share them all. If you like this issue, please let us know and maybe we can start doing an annual Publisher's Choice.

This is a good opportunity to give a special thanks to all the wonderful authors who support CrossTalk. There are many great people involved with putting together CrossTalk each month, and this journal's quality would be greatly reduced without their continued support. However, we wouldn't have a journal at all without the continued support of our authors. These people receive no compensation from CrossTalk except our thanks and a few extra copies of that month's issue, yet I believe the reward to our readers is great.

I always appreciate the shared knowledge from our authors, and it is a pleasure working with them. One example is an author that is one of the top 10 software experts in the United States, yet always acts as if I am doing *him* a favor when I publish one of his articles. Another example is an author that absolutely captivates me whenever I read her articles or listen to her speak. I was foolish enough to think she was CrossTalk's own special find until I noticed that she was the keynote speaker at a prestigious software conference in Europe. Then there are all the authors that I enjoy having an excuse to call and talk to for a few minutes about their article since they are just pleasant people.

We start this issue with an article that has been waiting way too long to be shared. Tom Gilb provides his insights for developing quality software in *The 10 Most Powerful Principles for Quality in Software and Software Organizations*. I hope he forgives our delay in sharing this information and will consider writing for us again.

Next is the second part of Alistair Cockburn's article, *Learning From Agile Software Development – Part Two*. Cockburn discusses the final three of 10 principles that are useful for setting up and running projects. He then concludes by discussing how plan-driven projects can borrow from agile methodologies.

Thomas C. Staab shares his experience with the Software Testing Maturity Model (SW-TMM) and how to use it either alone or in conjunction with the Capability Maturity Model for Software in his article, *Using SW-TMM to Improve the Testing Process*. Donald E. Casavecchia shares some practical approaches to configuration management (CM) in *Reality Configuration Management*. In this article, Casavecchia discusses how his organization has successfully used varying amounts of CM, depending on the needs of a project.

Gregory T. Daich's article, *Document Diseases and Software Malpractice*, provides a tongue-in-cheek discussion of some common "diseases" plaguing software and developers. Also, Capers Jones provides insight into the current state of software development within the U.S. military in *Defense Software Development in Evolution*.

I don't usually discuss BackTalk articles in this column, but as I stated earlier, this month was especially fun for me. After last May's BackTalk by Dr. David Cook, my brother e-mailed a rebuttal listing several counterpoints. I enjoyed reading them and asked him to expand them into an article for us. The result can be found in Kevin Leachman's *Trials and Tribulations of a Non-Geek Engineer*. (You know our parents will be receiving an autographed copy of this month's issue.)

Thank you again to all of the authors who contribute to CrossTalk. We work hard to share your ideas with our readers and hope you will continue to make them available.

Elizabeth Starrett
Associate Publisher

The 10 Most Powerful Principles for Quality in Software and Software Organizations

Tom Gilb
Result Planning Limited

The software industry knows it has a problem: The industry's maturity level with respect to "numbers" is known to be poor. While solutions abound, knowing which solutions work is the big question. What are the most fundamental underlying principles in successful projects? What can be done right now? The first step is to recognize that all your quality requirements can and should be specified numerically. This does not mean "counting bugs." It means quantifying qualities such as security, portability, adaptability, maintainability, robustness, usability, reliability, and performance. This article presents 10 powerful principles to improve quality that are not widely taught or appreciated. They are based on ideas of measurement, quantification, and feedback.

All projects have some degree of failure compared with initial plans and promises. Far too many software projects fail totally. In the mid 1990s, the U.S. Department of Defense (DoD) estimated that about half of its software projects were total failures [1]. The civil sector is no better [2]. So what can be done to improve project success? This article outlines 10 key principles of successful software development methods that characterize best practices.

These 10 principles have been selected because there is practical experience showing that they really gain control over qualities and their costs. They have a real track record spanning decades of practice in companies like IBM, Hewlett Packard, and Raytheon. They are not new: They are classic. But the majority of our community is young and experientially new to the game, so my job is to remind the industry of the things that work well. Your job is to evaluate this information and start getting the improvements that your management wants in terms of quality and the time and effort needed to get them.

"Those who do not learn from history, are doomed to repeat it" [3].

Principle 1: Use Feedback

The practice of gaining experience from formal feedback methods is decades old, and many appreciate its power. However, far too many software engineers and their managers are still practicing low feedback methods, such as waterfall project management (also known as Big Bang or Grand Design). Even many textbooks and courses continue to present low feedback methods. This is not done in conscious rejection of high feedback methods but from ignorance of the many successful and well-documented projects that have detailed the value of high feedback methods.

Methods using feedback succeed; those without feedback seem to fail. Feedback is the single most powerful principle for software engineering. (Most of the other principles in this article support the use of feedback.) Feedback helps you get better control of your project by providing facts about how things are working in practice. Of course, the presumption is that the feedback comes early enough to do some good; rapid feedback is the crux. We need to have the project time to make use of the feedback (for example, to radically change direction, if that is necessary). Four of the most notable rapid high-feedback methods are discussed in the following sections:

> *"The key feedback idea is to decentralize the initial causal analysis activity by investigating defects back to the grassroots programmers and analysts."*

Defect Prevention Process

The Defect Prevention Process (DPP) equates to the Software Engineering Institute's Capability Maturity Model® (CMM®) Level 5 as practiced at IBM from 1983 to the present [4]. The DPP is a successful way to remove the root causes of defects. In the short term (one year) about a 50 percent defect reduction can be expected; within two to three years, about a 70 percent reduction (compared to the original level) can be experienced; and in five to eight years, about a 95 percent defect reduction is possible [5].

The key feedback idea is to *decentralize* the initial causal analysis activity by investigating defects back to the grassroots programmers and analysts. This gives you the true causes and acceptable, realistic change suggestions. Deeper *cause analysis* and *measured process-correction* work can then be undertaken outside of deadline-driven projects by the more specialized and centralized process improvement teams.

There are many feedback mechanisms. For example, same-day feedback is obtained from the people working with the specification, and early numeric process change-result feedback is obtained from the process improvement teams.

Inspection Method

The Inspection Method originated at IBM in work carried out by M. Fagan, H. Mills (cleanroom method), and R. Radice (CMM inventor) [6]. Originally, it primarily focused on bug removal in code and code-design documents. Many continue to use it this way today. However, inspection has changed character in recent years. Today, it can be used more cost-effectively by focusing on measuring the significant defects on upstream specifications. Furthermore, sample areas often only need to be inspected rather than processing the entire document [7]. For example, the defect level measurement should be used to decide whether the entire specification is fit for release downstream to be used for a *go/no-go* decision-making review or for further refinement (test planning, design, or coding).

The main Inspection Method feedback components are as follows:

- Feedback to author from colleagues

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

regarding compliance with software standards.
- Feedback to author about required levels of standards compliance in order to consider their work releasable.

Evolutionary Project Management

Evolutionary Project Management (Evo, which originated in large scale within cleanroom methods) has been successfully used on the most demanding space and military projects since 1970 [8, 9]. The DoD changed its software engineering standard MIL-STD-2167A to an Evo standard (MIL-STD-498), which derived succeeding public standards, (for example, the Institute of Electrical and Electronics Engineers). The reports, (op. cit.) along with my own experience, are that Evo results in a remarkable ability to deliver on time and on budget, or better, compared to conventional project management methods [2].

An Evo project is consciously divided into small, early, and frequently delivered stakeholder result-focused steps. Each step delivers benefits and builds toward satisfaction of the final requirements. Step size is typically weekly or 2 percent of total time or budget. This results in excellent regular and realistic feedback about the team's ability to deliver meaningful, measurable results to selected stakeholders. The feedback includes information on design suitability, stakeholders' reactions, requirements' trade-offs, cost estimation, time estimation, people resource estimation, and development process aspects.

Statistical Process Control

Statistical Process Control [10], although widely used in manufacturing [11], is only used in software work to a limited degree. Some use is found in advanced inspections [5, 12]. The Plan Do Study Act cycle is widely appreciated as a fundamental feedback mechanism.

Principle 2: Identify Critical Measures

It is true of any system – your body, an organization, a project, software, or service product – that there are several factors that can cause a system to die. Managers call these *critical success factors*. If you analyzed systems looking for all the critical factors that cause shortfalls or failures, you would get a list of factors needing better control. They would include both stakeholder values (such as serviceability, reliability, adaptability, portability, and usability) and the critical resources needed to deliver those values (i.e., people, time, money, and data quality). For each critical factor, you would find a series of faults that would include the following:

- Failure to systematically identify all critical stakeholders and their critical needs.
- Failure to define the factor measurably. Typically, only buzzwords are used and no indication is given of the survival (failure) and target (success) measures.
- Failure to define a practical way to measure the factor.
- Failure to contract measurably for the critical factor.
- Failure to design toward reaching the factor's critical levels.
- Failure to make the entire project team aware of the numeric levels needed for the critical factors.
- Failure to maintain critical levels of performance during peak loads or on system growth.

Our entire culture and literature of *software requirements* systematically fails to account for the majority of critical factors. Usually, only a handful such as performance, financial budget, and deadline dates are specified. Most quality factors are not defined quantitatively at all. In practice, all critical measures should always be defined with a useful scale of measure. However, people are not trained to do this and managers are no exception. The result is that our ability to define critical *breakdown* levels of performance and manage successful delivery is destroyed from the outset.

	Step #1 Plan A: {Design: X, Function: -Y}	Step #1 Actual	Step #1 Difference - Is Bad + Is Good	Total Step #1	Step #2 Plan B: {Design: Z, Design: F}	Step #2 Actual	Step #2 Difference	Total Steps #1 and #2	Step #3 Next Step Plan
Reliability 99%-99.9%	50% ±50%	40%	-10%	40%	30% ±20%	20%	-10%	60%	0%
Performance 11 sec.-1 sec.	80% ±40%	40%	-40%	40%	30% ±50%	30%	0	70%	30%
Usability 30 min.-30 sec.	10% ±20%	12%	+2%	12%	20% ±15%	5%	-15%	17%	83%
Capital Cost 1 mill.	20% ± 1%	10%	+10%	10%	5% ± 2%	10%	-5%	20%	5%
Engineering Hours 10,000	2% ± 1%	4%	-2%	4%	10% ± 2.5%	3%	+7%	7%	5%
Calendar Time	1 week	2 weeks	-1 week	2 weeks	1 week	0.5 week	+0.5 week	2.5 weeks	1 week

Table 1: *Example of an Impact Estimation Table*

Principle 3: Control Multiple Objectives

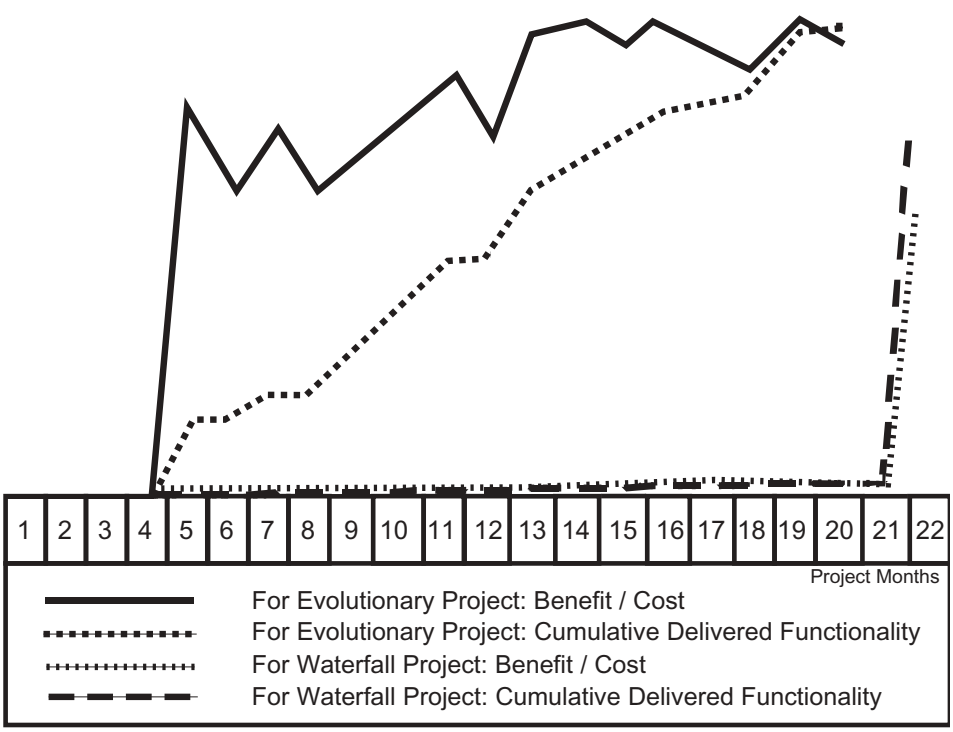
You do not have the luxury of managing qualities and costs at whim. With software development, you cannot decide to manage just a few of the critical factors and avoid dealing with the others. You have to deal with *all* the potential threats to your project, organization, or system. You must simultaneously track and manage all the critical factors. If not, then the *forgotten factors* will probably be the very reasons for project or system failure.

I have developed the Impact Estimation (IE) method (see Table 1) to enable tracking of critical factors; however, it does require that critical objectives and quantitative goals have been identified and specified. Given that most software engineers have not yet learned to specify *all* their critical factors *quantitatively* (Principle 2), this *next* step, tracking progress against quantitative goals to enable control of multiple objectives (this principle), is usually impossible.

IE is conceptually similar to Quality Function Deployment [13], but it is much more objective and numeric. It gives a picture of reality that can be monitored [14, 15] (Table 1). It is beyond the scope of this article to provide all the underlying detail for IE. To give a brief outline, the percentage estimates in Table 1 are based, as far as possible, on source-quoted, credibility-evaluated, objective, documented evidence. IE can be used to evaluate ideas *before* their application, and it can also be used, as in Table 1, to track progress toward multiple objectives *during* an evolutionary project. In Table 1, the *Actual Difference* and *Total* numbers represent *feedback* in small steps for the chosen set of critical factors that management has decided to monitor. If the project is deviating from plans, this will be easily visible and can be corrected in the next step.

Principle 4: Evolve in Small Steps

Software engineering is by nature playing with the unknown. If we already had exactly what we needed, we would reuse it. When we choose to develop software, there are many types of risk that threaten



Note: One advantage of Evo is that you can focus on delivering high value increments to critical stakeholders early. The upper line represents high value at early stages [17].

Figure 1: *Evolutionary vs. Waterfall Comparison*

the result. One way to deal with this is to tackle development in small steps, one step at a time. If something goes wrong, we will immediately know it. We also have the ability to retreat to the previous step, a level of satisfactory quality, until we understand how to progress again.

It is important to note that the small steps are not mere development increments. The point is that they incrementally satisfy identified stakeholder requirements (see Figure 1). Early stakeholders might be salespeople needing a working system for demonstration, system installers/help desk/service/testers who need to work with something, or early trial users.

The duration of each small step is typically a week or so. The smallest widely reported steps are the daily builds used at Microsoft, which are useful-quality systems. They cumulate to six- to 10-week *shippable quality* milestones [16].

Principle 5: A Stitch in Time Saves Nine

Quality control must be done as early as possible, from the earliest planning stages, to reduce the delays caused by finding defects later. There needs to be strong specification standards (such as *all quality requirements must be quantified*) and rigorous checking to measure that the rules are applied in practice. When the specifications are not of some minimum standard (like "<1 major defect/page remaining") then they must be edited until they become acceptable, including the following:

- Use inspection sampling to keep costs down, and to permit early, i.e., before specification completion, correction and learning.
- Use numeric exit from development processes such as *Maximum 0.2 Majors per page*.

It is important that quality control by inspection be done very early for large specifications, for example within the first 10 pages of work. If the work is not up to standard, then the process can be corrected before more effort is wasted. I have seen half a day of inspection (based on a random sample of three pages) show that there were about 19 logic defects per page in 40,000 pages of air traffic control logic design. The same managers who had originally *approved* the logic design for coding carried out the inspection with my help. Needless to say, the project was seriously late.

In another case I facilitated (United States, 1999, jet parts supplier), eight managers sampled two pages out of an 82-page requirements document and measured 150 *major* defects per page. Unfortunately, they had failed to do such sampling three years earlier when the project started, so they had already experienced one year of delay; they told me they expected another year delay while removing the injected defects from the project. This two-year delay was accurately predictable given the defect density they found and the known average cost from major defects. They were amazed at this insight, but agreed with the facts. In theory, they could have saved two project years by doing early quality control against simple standards: clarity, unambiguity, and no design in requirements.

These are not unusual cases. I find them consistently all over the world. Management frequently allows extremely weak specifications to go unchecked into costly project processes. They are obviously not managing properly.

Principle 6: Motivation Moves Mountains

Motivation is everything! When individuals and groups are not motivated positively, they will not move forward. When they are negatively motivated (fear, distrust, and suspicion), they will resist change to new and better methods. Motivation is a type of method. In fact, there are many large and small items contributing to your group's *sum of motivation*. We can usefully divide the *motivation problem* into four categories:

- The will to change.
- The knowledge to change direction.
- The ability to change.
- The feedback about progress in the desired change direction.

Leaders (I did not say managers) create the will to change by giving people a positive and fun challenge and the freedom and resources to succeed. During the 1980s, John Young, CEO of Hewlett Packard, inspired his troops by saying that he thought they needed to aim to be measurably 10 times better in service and product qualities by the end of the decade. He did not demand it. He supported them in doing it. They reported getting about 9.95 times better, on average, in the decade. The company was healthy and competitive during a terrible time for many others.

The knowledge of directional change is critical to motivation; people need to channel their energies in the right direction! In the software and systems world, this problem has three elements, two of which have been discussed in earlier principles. They are as follows:

- Measurable, quantified clarity of the requirements and objectives of the various stakeholders (Principle 2).
- Knowledge of all the multiple critical goals (Principle 3).
- Formal awareness of constraints such as resources and laws.

These elements are a constant communication problem because of the following:

- We do not systematically convert our directional changes into crystal clear measurable ideas; people are unclear about the goals and there is no ability to obtain numeric feedback about movement in the *right* direction. We are likely to say we need a *robust* or *secure* system, and less likely to convert these rough ideals into concrete, measurable, defined, agreed-upon requirements or objectives.
- We focus too often on a single measurable factor (such as *percent built* or *budget spent*) when reality demands that we simultaneously track multiple critical factors to avoid failure and to ensure success. We do not understand what we should be tracking, and we do not get enough *rich* feedback.

Principle 7: Competition Is Eternal

Our conventional project management ideas strongly suggest that projects have a clear beginning and a clear ending. In our competitive world, this is not as wise a philosophy as one W. Edwards Deming suggests, "Eternal process improvement is necessary as long as you are in competition" [11]. We can have an infinite set of *milestones* or evolutionary steps of result delivery and use them as we need; the moment we abandon a project, we hand opportunity to our competitors. They can sail past our levels of performance and take our markets.

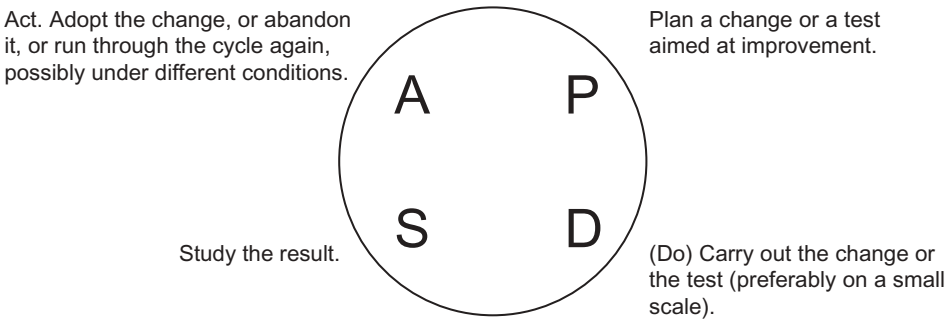
The practical consequence is that our entire mindset must always be on setting new ambitious numeric *stakeholder value* targets both for our organizational capability and our product and service capabilities (see Figure 2).

Continuous improvement efforts in the software and services area at IBM, Raytheon, and others [4, 5, 18] show that we can improve critical cost and performance factors by 20 to one, in five- to eight-year time frames. Projects must become *eternal* campaigns to get and stay ahead.

Principle 8: Things Take Time

"It takes two to three years to change a project, and a generation to change a culture" [11].

Technical management needs to have a long-term plan for improving the critical characteristics of their organization and their products. Such long-term plans need the ability to be tracked numerically and stated in multiple critical dimensions. At the same time, visible short-term progress toward those long-term goals should be planned, expected, and tracked (see Figure 3).



Note: Reproduction from a letter from W. Edwards Deming, May 18, 1991 to the author.

Figure 2: *The Shewhart Cycle for Learning and Improvement – the PDSA Cycle*

Principle 9: The Bad With the Good

Any method (means, solution, or design) you choose will have multiple quality and cost impacts whether you like them or not! In order to get a correct picture of how good any idea is for meeting our purposes, we must do the following:

- Have a quantified, multidimensional specification of our requirements, our quality objectives, and our resources (people, time, or money).
- Have knowledge of the expected impact of each design idea on all these quality objectives and resources.
- Evaluate each design idea with respect to its total – expected or real – impact on our requirements, the unmet objectives, and the unused cost budgets.

We need to estimate all impacts on our objectives. We need to reduce, avoid, or accept negative impacts. We must avoid simplistic one-dimensional arguments. If we fail to use this systems engineering discipline, then we will be met with unpleasant surprises of delays and bad quality, which seem to be the norm in software engineering today. One practical way to model these impacts is using an IE table (see Table 1, page 5).

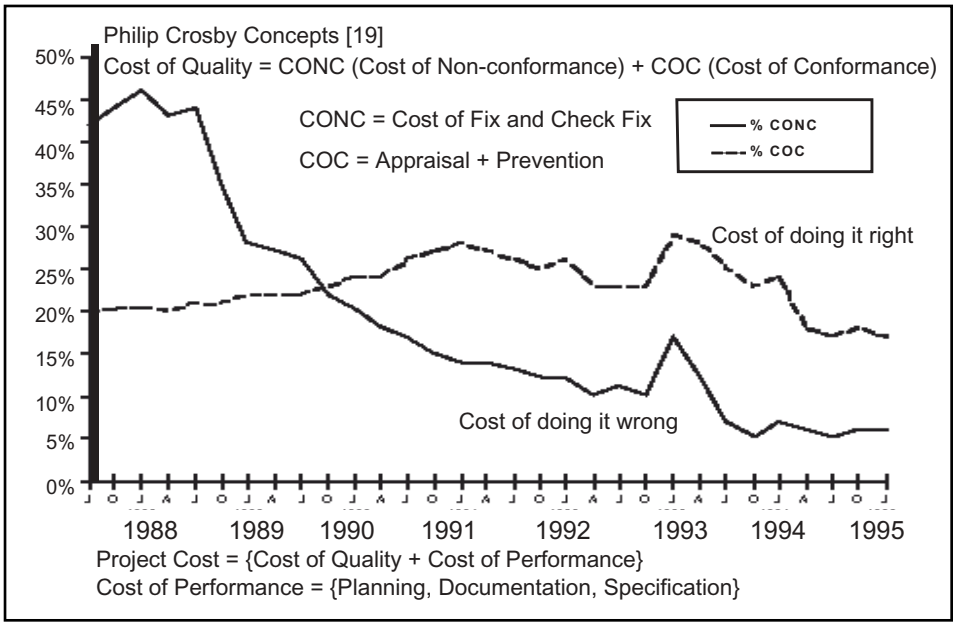
Principle 10: Keep Your Eyes on Where You Are Going

"Perfection of means and confusion of ends seem to characterize our age," said Albert Einstein.

To discover the *real* problem, we have only to ask of a specification: Why? The answer will be a higher level of specification, nearer the real ends. There are too many designs in our requirements!

You might say, why bother? Isn't the whole point of software to get the code written? Who needs high-level abstractions? Cut the code! But somehow that code is late and of unsatisfactory quality.

Figure 3: *Cost of Quality vs. Time: Raytheon 95 – the Eight-Year Evolution of Rework Reduction*



Note: In the case of Raytheon process improvements (Dion, 1995), many years of persistent process change for 1,000 programmers were necessary to drop rework costs from 43% of total software development costs to below 5%.

COMING EVENTS

December 9-11
7th Annual ITC East Conference
Hershey, PA
www.govresources.com

January 14-16, 2003
West 2003 Conference
San Diego, CA
www.west2003.org

February 24-27
Software Engineering Process Group Conference

Boston, MA
www.sei.cmu.edu/sepg/

April 7-11
Software Development Conference and Expo West
Santa Clara, CA
www.sdexpo.com/2003/west

April 8-10
FOSE 2003 Conference
Washington, DC
www.fose.com

April 28-May 1
Software Technology Conference 2003

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

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

May 6-8
TechNet International 2003
Washington, DC
www.technet2003.org

June 25-28
Agile Development Conference
Salt Lake City, UT
www.agiledevelopmentconference.com

The reason is often lack of attention to the real needs of the stakeholders and the project. We need these high-level abstractions of what our stakeholders need so that we can focus on giving them what they are paying us for! Our task is to design and deliver the best technology to satisfy their needs at a competitive cost.

One day, software engineers will realize that the primary task is to satisfy their stakeholders. They will learn to design toward stakeholder requirements (multiple simultaneous requirements). One day we will become real systems engineers and realize there is far more to software engineering than writing code.

Conclusion

Motivate people toward real results by giving them numeric feedback frequently and the freedom to use any solution that gives those results. It is that simple to specify. It is that difficult to do.◆

References

1. Jarzombek, Stanley J. "The 5th Annual Joint Aerospace Weapons Systems Support, Sensors, and Simulation Symposium (JAWS S3)." Proceedings, 1999.
2. Morris, Peter W. G. The Management of Projects. Ed. Thomas Telford. London, 1994.
3. Santayana, George. The Life of Reason. Amherst: Prometheus Books, 1903.
4. Mays, Robert. Practical Aspects of the Defect Prevention Process. (Gilb, Tom, and Dorothy Graham. Software Inspection. Addison-Wesley, 1993. Chapter 17 written by Mays).
5. Dion, Raymond, et. al. The Raytheon Report. Pittsburgh: Software Engineering Institute, 1995 <www.sei.cmu.edu/publications/documents/95.reports/95.tr.017. html>.
6. Fagan, Michael E. "Design and Code Inspections." IBM Systems Journal 15.3 (1976): 182-211. Reprinted 38.2, 3 (1999): 259-287 <www.almaden.ibm.com/journal>.
7. Gilb, Tom, and Dorothy Graham. Software Inspection. Addison-Wesley, 1993. Japanese Translation, Aug. 1999.
8. Mills, Harlan D. IBM Systems Journal. 1980. Also republished IBM Systems Journal, Nos. 2 and 3, 1999.
9. Cotton, Todd. "Evolutionary Fusion: A Customer-Oriented Incremental Life Cycle for Fusion." Hewlett-Packard Journal 47.4 (Aug. 1996): 25-38.
10. Shewhart, Deming, Juran 1920s.
11. Deming, W. Edwards. Out of the Crisis. Cambridge: MIT CAES Center for Advanced Engineering Study, 1986.
12. Florac, William A., Robert E. Park, and Anita D. Carleton. Practical Software Measurement: Measuring for Process Management and Improvement. Pittsburgh: Software Engineering Institute, 1997 <www.sei.cmu.edu>.
13. Akao, Yoji. Quality Function Deployment: Integrating Customer Requirements into Product Design. Cambridge: Productivity Press, 1990.
14. Gilb, Tom. Principles of Software Engineering Management. Boston: Addison-Wesley, 1988.
15. Gilb, Tom. Competitive Engineering. Addison-Wesley: United Kingdom, 2000 <www.resultplanning.com>.
16. Cusumano, Michael A., and Richard W. Selby. Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People. The Free Press (a division of Simon and Schuster), 1995.
17. Woodward, Stuart. "Evolutionary Project Management." IEEE Computer Oct. 1999: 49-57.
18. Kaplan, Craig, Ralph Clark, and Victor Tang. Secrets of Software Quality, 40 Innovations From IBM. McGraw Hill, 1944.
19. Crosby, Philip B. Quality Is Still Free: Making Quality Certain in Uncertain Times. McGraw Hill, 1996.

About the Author

Tom Gilb has been a freelance consultant since 1960 and is the author of nine books, including "Software Metrics," "Principles of Software Engineering Management," "Software Inspection," and the forthcoming "Competitive Engineering." Gilb teaches and consults worldwide with major multinational clients including Nokia, Ericsson, Motorola, HP, IBM, BAE Systems, Philips, Sony, Canon, Intel, and Microsoft and does pro bono training and consulting for the Department of Defense, United Kingdom, NATO, and the Norwegian Defense.

Iver Holtersvei 2
NO-1410
Kolbotn, Norway
Phone: +47 66 80 46 88
E-mail: tom@gilb.com

Learning From Agile Software Development – Part Two

Alistair Cockburn
Humans and Technology

This two-part article compares agile, plan-driven, and cost-sensitive software development approaches based on a set of project organization principles, extracting from them ideas for pulling agile techniques into cost- and plan-driven projects. Part one, which appeared in October's CrossTalk, *described how the different teams make trade-offs of money for information or for flexibility, and presented the first seven of 10 principles for tuning a project to meet various prioritization of cost, correctness, predictability, speed, and agility. This month, part two presents the last three principles, then pulls the material together for actions that plan-driven and cost-sensitive project teams can use to improve their strategies and hedge against surprises.*

Being agile is a declaration of priorities, prioritizing for project maneuverability with respect to shifting requirements, shifting technology, and shifting understanding of the situation. Other priorities that might override agility include predictability, cost, schedule, process-accreditation, or use of specific tools.

Most managers run a portfolio of projects having a mix of those priorities and need to mix their strategies to suit. The question at hand is how a person can borrow from the set of agile practices to fit the plan-driven and cost-sensitive programs.

Part one of this article [1] introduced two phrases:

1. *Money-for-information* (MFI) issues are those on which the team can spend money now to obtain information that puts them in a better situation for later in the project. Work-plan breakdown structures, system performance under load, and user reaction to system design are MFI issues.
2. *Money-for-flexibility* (MFF) issues are those on which the team cannot possibly obtain information now to put them in a better situation later. The better strategy is to spend money on making the change easier later. Movements in the stock market, emerging standards, and staff continuity are MFF issues.

Many of the differences between agile methodologies and cost- and plan-driven approaches are in deciding which issues are MFF or MFI issues, and what the best allocation of resource is for each.

A plan-driven team might decide that the project plan is predictable, and a good MFI strategy is to spend energy now to make those predictions. An agile team might decide that the project plan fundamentally cannot be resolved past a very simple approximation, and therefore a MFI strategy is a waste of money. Instead, they adopt a MFF approach, which involves making many coarse-grained plans over the course of the project.

Both teams might agree that the question of system performance under load is an important MFI issue, and both might agree to spend money early to build a simple system simulator and load generator to stress test the design.

> *"The most important agile value for the cost-sensitive project leader to adopt is customer collaboration."*

Ten Principles

The following 10 principles have shown themselves useful in setting up and running projects:

1. Different projects need different methodology trade-offs.
2. A little methodology does a lot of good; after that, weight is costly.
3. Larger teams need more communication elements.
4. Projects dealing with greater potential damage need more validation elements.
5. Formality, process, and documentation are not substitutes for discipline, skill, and understanding.
6. Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.
7. Increasing feedback and communication reduces the need for intermediate work products.
8. Concurrent and serial development exchange development cost for speed and flexibility.
9. Efficiency is expendable in non-bottleneck activities.
10. Sweet spots speed development.

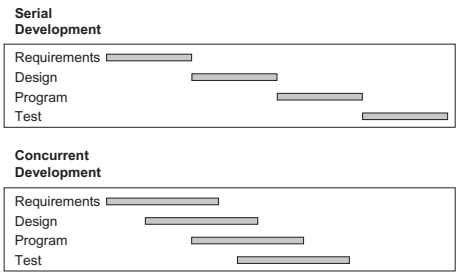
The first seven principles were addressed in the October issue of CrossTalk. I pick up the discussion here with Principle No. 8.

8. Concurrent and Serial Development Exchange Development Cost for Speed and Flexibility

On a predictable project, the project coordinator can arrange for each work specialist to show up at just the right moment, perform the needed work, and leave. Such scheduling, common in the construction and book publishing industries, minimizes salary cost in exchange for extending elapsed time (see Figure 1, Serial Development). The hazard is that a surprise might show up in an already-completed task forcing the previous task item to restart, in which case neither time nor cost is minimized.

Concurrent development runs teams in parallel, even when they have dependencies between them [2]. The teams will make and change decisions as they gain information, causing the other teams some rework. With careful management of their dependencies, the teams can complete the

Figure 1: *Serial Development vs. Concurrent Development* [2]



Note: Serialized development takes longer but costs less than concurrent development.

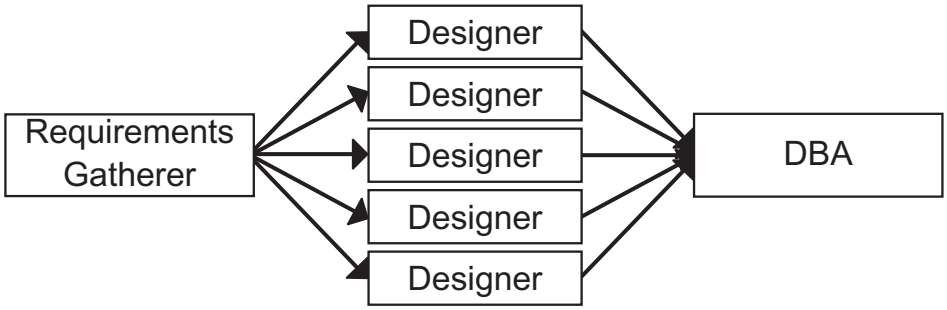


Figure 2: *The Database Analyst (DBA) Bottlenecking Five Designers* [2]

final work sooner even though their salary costs are higher (see Figure 1, Concurrent Development). Effective concurrent development demands that communication between people is fast, rich, and inexpensive (as discussed in principles 6 and 7). The hazard in concurrent development is that if work is started too early, rework costs dominate the project.

Serial and concurrent development have opposing characteristics. Cost-sensitive projects should use serial development where they can, while projects sensitive to shifting requirements benefit more from concurrent development

Agile project teams almost always use concurrent development assuming a significant number of surprises will arise during development. The close communication needed for effective concurrent development also lets them respond to late-breaking changes effectively.

9. Efficiency Is Expendable in Non-Bottleneck Activities

Effective concurrent development requires calculating the moment at which to start a downstream activity. Goldratt's process theory [3] and theory of constraints [4] provide advice here.

Suppose that one requirements gatherer feeds information to five designers, who in turn feed their results to a single database analyst (DBA, see Figure 2). It is clear that the DBA will not be able to keep up with the work (and rework). Prudence insists that the designers get their work to a complete and stable state before passing it to the DBA.

Figure 3 illustrates this idea. The vertical axis indicates how complete and stable each group's work is. Completeness refers to how much they have done, and stability refers to how unlikely they are to make changes. For simplicity, the figure illustrates them as joint: The work becomes more complete and more stable over time, shown in an S-type of curve. For each curve, the solid downward-arrow indicates at what point a dependent activity gets initiated.

If the designers take work from a single requirements gatherer as in Figure 2, they can start work on their assignments when the requirements are only slightly complete and stable and still handle the consequential rework. Figure 3 shows the trigger event (the solid vertical arrow from requirements to design) occurring close to the left, while the requirements are not yet very complete or very stable. This figure also shows information continuing to pass from the requirements gatherer to the designers as the requirements work progresses. Once requirements become complete and stable, it will not take long to finalize work. The designers can complete the extra rework because there are five of them to one requirements gatherer.

The DBA, having no excess capacity, needs to be handed work that is more complete and more stable. The solid rightmost vertical arrow in Figure 3, which shows when the DBA's work gets initiated, starts higher on the designer's completeness and stability scale.

Note that in Figure 3 each designer uses much more time than the DBA. This is appropriate, since there is only one DBA for five designers.

The principle says that rework is an expendable commodity everywhere except at the bottleneck station (the DBA, in the above example). Rework can be expended to improve a design, to investigate multiple designs, or to get a head start on a downstream activity. Applying this principle to different circumstances produces different optimal project strategies [2].

Although this is the most complicated principle presented so far, I find that most project leaders have, in fact, used this principle in responding to standard project pressures through common sense and intuition.

10. Sweet Spots Speed Development

The ideal project uses dedicated, experienced people who sit within earshot of each other; use automated regression tests; have easy access to the users; and deliver running, tested systems to those users every month or two. Such a project is clearly in a better position to complete successfully than one missing those characteristics. The surprise is that sponsoring executives do not pay more attention to these important success factors.

When the team cannot hit one of those sweet spots, then they need to invent a way to get closer to it. The farther away they are, the more difficult the project becomes. Here are six sweet spots:

1. **Dedicated Developers**. There is a large emotional and mental cost to a person having to switch between multiple assignments [2, 5]. In my project reviews, I find that once people get interrupted at the rate of about three times per day, they stop even trying to focus on their main assignment and simply wait for the next interruption to happen. One senior project manager reported that he simply does not count as productive staff anyone assigned less than half-time to the project.
2. **Experienced Developers**. Experienced developers know the domain, they know the technologies or how to adopt them, and they know their computer science material. They move at multiple times the speed of their slower colleagues.
3. **Small Collocated Team** (a consequence of principles 6 and 7). Two to eight people sitting in the same room can ask each other questions without raising their voices. They are aware of when others are available to answer questions. They overhear relevant conversations without pausing in their work. They keep the design ideas and project plans on the board in ready sight and share information faster. The developers I have interviewed uniformly say that while the environment can get noisy, they have never been on a more effective project than when a small team sat in the same room.

   Technology can mitigate the situation somewhat. One project team installed cameras on every workstation to display the image of the other people on the project in their various offices [6]. This gave them a sense of each other's presence, and indicated when people were not at their workstations or not to be disturbed with a question. They used online chat boxes to fire off and get answers to the many small questions that constantly arise. They were creative in mimicking the sweet spot in an otherwise unsweet situation.
4. **Automated Regression Tests**. With

automated regression unit and acceptance tests, the developers can revise the code base and retest the entire system at the push of a button. Teams who have such tests report that they freely replace and improve awkward modules. They also report relaxing more on the weekends since they will run the tests on Monday morning and discover if someone has changed their system out from under them. These tests improve both the design quality and the programmers' quality of life.

Experienced developers spend quite some effort to minimize the amount of the system not amenable to automated regression tests.

5. **Easy Access to Users**. Having a *customer* or usage expert available at all times means that the feedback cycle from nominated solution to evaluated idea is much shorter, often in the range of minutes to a few hours. The development team gains a deeper understanding of the users' needs and habits and makes fewer mistakes nominating ideas. It also means that more ideas can be tried, allowing for a better final product.

   Missing this sweet spot lowers the likelihood of making a really useable product. Teams unable to have a usage expert available at all times have substituted weekly sessions with the users, studying the user community in depth before and during the project, using surveys, or using friendly alpha-test groups.
6. **Short Increments and Frequent Delivery to Real Users**. There is no substitute for rapid feedback, both on the development process and the product itself. Some colleagues say that even one month is an intolerably long time. However, there is also a cost to deploying a product, which makes this a MFI proposition (discussed in the previous article).

   With short increments, the process itself gets tested and can be repaired quickly, and the requirements for the product can be tested and varied quickly.

   Projects that cannot deliver to an end user every few months should integrate a full build every few months and pretend as though it were delivered. This way, they exercise every part of the development process.

Differences Between Approaches

At this point, we have listed the issues that bear on how cost- and plan-driven projects can borrow from agile approaches. Some cause intrinsically different responses; other responses are more a matter of habit.

Intrinsic Differences

Statistics vs. heuristics. Some project leaders believe software development is a statistically controllable process; others do not. Their resulting strategies are incompatible. This is one of the places where friction arises between agile and plan-driven project leaders.

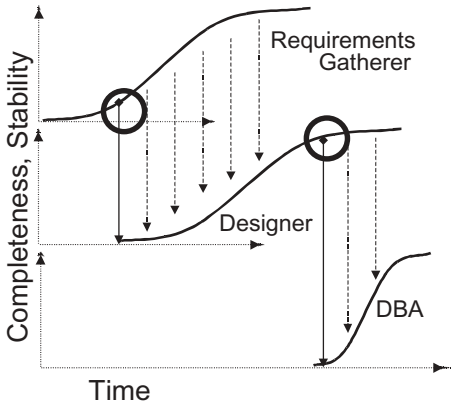
Individuals and interactions vs. processes and tools. Some leaders believe that with the right process, they can become immune to the turnover of key people. Others believe that no process can offer that immunity; the heart of good software development will always reside in the individual people on the project. As with the statistical approach, we are more likely to find process-centric leaders running plan-driven projects, and individual-centric leaders running agile projects.

Responding to change vs. following a plan. It is a fundamental difference between the two project types whether the team is encouraged to or penalized for responding to changes. Even though business needs, requirements, technologies, and people are constantly moving these days, some projects are still fixed in some combination of time, scope, and cost, and must operate in the plan-driven range.

Project plan as MFI or MFF. If the requirements or technologies are likely to change late in the game or without notice, or the team does not have experience with the technology, then it is a poor strategy to treat the project plan as a MFI issue. In those situations, the agile leader's mindset that the plan is a MFF proposition works better. The leader allocates energy to replanning coarsely but frequently.

Design as MFI or MFF. A plan-driven project team, believing that the design can be worked out in advance (MFI), expends resources early to gain that information and lock down the design. For those design elements that cannot be foreseen (MFF issues), plan-oriented design teams often design the system so that a range of future design constraints can be easily incorporated – expending extra design energy early in anticipation of having a more adaptable design.

Many agile designers find those resulting designs overly complicated. Agreeing that certain issues are MFF issues, they argue that a better MFF strategy is to make a simpler design in the first place, with less built-in flexibility. The saved money can then be allocated to change the design on an as-needed basis.



Note: The designer-programmer benefits the schedule by starting earlier and accepting more rework [2].

Figure 3: *Completeness and Stability Over Time*

Some agile designers argue that the MFI component of the design activity is negligibly small, thus little or no effort should be expended on anticipated design changes.

Serial vs. concurrent development. There is a fundamental difference in the strategies applied when agility is a priority compared with when cost is the priority. As Principle No. 8 describes, cost-sensitive projects do better with serial development when that can be successfully executed. Unfortunately, there are so many surprises in projects that it is very difficult to execute successfully.

Surmountable Differences

Working software vs. comprehensive documentation. One tends to find more initiatives for comprehensive documentation on statistics-, process- and plan-driven projects, but this is not intrinsic. Many experienced managers use prototypes, simulators, and incremental development to reduce risk and gain early information on both agile and plan-driven projects, feeding that information into the plan as quickly as possible.

Customer collaboration vs. contract negotiation. Plan-driven project leaders clearly can improve their situation by increasing the collaboration in their customer relations, even if they must write and enforce contracts. This is a case in which plan-driven project leaders can employ some of the same work practices as agile project leaders.

Project plan and design on cost-sensitive projects. A detailed plan does not, by itself, confer cost savings or safety to a project. Detailed plans and detailed designs enable an estimable base-line cost. The manager can then tell if the cost is going up or down over time. It is not the

detail of the plan, but successful application of MFF and MFI decisions that makes the difference in the result.

Borrowing From Agile

Drawing from the above, we see that the plan-driven project can streamline its development operations, improve predictability, and hedge its bets by borrowing in various ways from the agile approach. Following are examples of these.

Streamlining

A plan-driven project leader should still try to hit the six sweet spots: dedicated, experienced and collocated staff; using automated regression test suites; having easy access to knowledgeable users; and showing and delivering incrementally growing, running, tested systems to them regularly.

In addition to these, the project members can question to what extent they can lower the documentation burden through a more informal information exchange.

Improving Predictability

Good project leaders already use prototypes, simulators, and incremental development to get early information on their project. However, in my experience, many leaders of plan-driven projects do not avail themselves of these techniques, which are standard business among agile developers.

Of the above techniques, the most important for the plan-driven team to adopt is incremental development. By delivering a few increments, the leader gains invaluable information about *this* team, *this* problem, and *this* technology. That data are more appropriate to the project plan than estimates from other people working on other problems in other technology.

Hedging Bets

Surprises can show up even on a plan-driven project. Based on where they estimate those surprises are, the plan-driven project leaders can incorporate some of the agile mindset into their strategy. Once again, the use of incremental development is key. The delivery, or even just integration, of each increment offers the team a chance to deal with whatever surprise showed up, whether in the requirements, the technology, or the process. The other technique to borrow is concurrent development, which offers a way to speed development and respond to late-breaking changes.

Lowering Costs

Customer collaboration over contract negotiation. The most important agile value for the cost-sensitive project leader to adopt is customer collaboration. When told that varying a requirement converts an expensive design into a simple, inexpensive one, a customer often is willing to change the requirements to allow the less expensive design. To the extent that the customer and the development team are on good terms, this happens more often.

Working software over comprehensive documentation. Tacit knowledge and informal communication are much less expensive than complete documentation. The cost-sensitive project will play a game of documentation brinkmanship, creating only minimal documents needed to keep the project from falling apart.

Responding to change vs. following a plan. Optimizing from an accurate plan is clearly a winning strategy. The only time that responding to change is advantageous to a cost-sensitive project team is when they discover a shortcut later in the project. At that point, they obviously benefit from changing the plan.

Summing Up

Agile teams put more emphasis on the ideas presented in this two-part article than do plan-driven teams. Most of the ideas are not particularly new. What is surprising is the extent to which these known, old practices are ignored. It is sobering to re-read the paper, "Disciplines Delivering Success," presented at the 1997 Software Technology Conference in Salt Lake City [7] in which Brown points out the following: "*project-saving disciplines ignored by management*: good personnel practices, planning and tracking using activity networks and earned value, incremental release build plan, formal configuration management, test planning and project stability, and metrics."

Of all the practices, the agile strategy of using concurrent development is intrinsically in opposition to cost-minimization under predictable circumstances. However, cost-sensitive project teams can benefit from all four of the agile values and all six of the project sweet spots. Customer collaboration and making good use of close, informal communications are key among those.

Of the differences between development styles, agile developers typically believe that software development is not amenable to statistical process control, and so heuristic project controls must be used.◆

References

1. Cockburn, A. "Learning From Agile Software Development – Part One." CrossTalk Oct. 2002: 10-14.
2. Cockburn, A. Agile Software Development. Boston: Addison-Wesley, 2001.
3. Goldratt, E. The Goal. Great Barrington: North River Press, 1992.
4. Goldratt, E. Theory of Constraints. Great Barrington: North River Press, 1990.
5. DeMarco, T., and T. Lister. Peopleware: Productive Projects and Teams. 2nd Ed. New York: Dorset House, 1999.
6. Herring, R., and M. Rees. Internet-Based Collaborative Software Development Using Microsoft Tools. Proceedings of the 5th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, Florida. July 2001: 22-25 <http://erwin.dstc.edu.au/Herring/SoftwareEngineering0verInternetSCI2001.pdf>.
7. Brown, N. "Disciplines Delivering Success." Software Technology Conference, 1997 <http://stc-online.org/cd-rom/1997/track1.pdf>.

About the Author

Alistair Cockburn, an internationally recognized expert in object technology, methodology, and project management, is a consulting fellow at Humans and Technology with more than 20 years experience. He is one of the original authors of the Agile Software Development Manifesto and founders of the AgileAlliance, and is program director for the Agile Development Conference held in Salt Lake City.

1814 Fort Douglas Circle
Salt Lake City, UT 84103
Phone: (801) 582-3162
Fax: (775) 416-6457
E-mail: alistair.cockburn@acm.org

Call for Conference Participation

The Agile Development Conference is seeking people to give tutorials, host workshops, or submit field reports or research papers at the conference June 25-28, 2003 in Salt Lake City. More information is available at <www.agiledevelopmentconference.com>.

Using SW-TMM to Improve the Testing Process

Thomas C. Staab
Wind Ridge International

Can using the Software Testing Maturity Model℠ (SW-TMM℠) really help improve your testing process? The answer is a resounding "Yes!" This article describes the SW-TMM and the benefits that can be derived from its use.

The Software Testing Maturity Model℠ (SW-TMM℠) is an exciting tool that can help generate significant changes in an organization's testing process. The SW-TMM is a testing process improvement tool that can either be used in conjunction with the Capability Maturity Model® for Software (SW-CMM®) or as a stand-alone tool. While the SW-CMM is an excellent software development tool that recognizes that reviews and testing are activities intended to enhance quality, it does not provide sufficient depth of testing coverage; the SW-TMM fills that void. The SW-TMM is not a tool to be used in addition to SW-CMM but is designed to be used in conjunction with the SW-CMM.

What Are Testing Maturity Models?

Testing maturity models are not new. Available documentation shows that most of them were developed around 1996, but they have never found much acceptance. One of the main reasons for this is the fact that there is very little documentation on the models. The articles, books, and Web sites on testing maturity models are written in a very theoretical style. Most people read them and say something like, "That's an interesting concept. When I have the time, I'll look into it." Of course, they never find the time.

I have taken the time to study the various maturity models and have discovered that the SW-TMM can be easily implemented and provides significant improvements in the testing process.

What Testing Maturity Model Should I Use?

Terry Weatherill with ImagoQA Ltd. undertook a comparison of the testing maturity models currently available. His article "In the Testing Maturity Model Maze" [1] documents the results of his comparison. He studied the following six testing maturity models:

- Testability Maturity Model [2].
- Software Testing Maturity Model.
- Test Process Improvement (TPI) [3].
- Test Organization Maturity™ [4].
- Testing Assessment Program [5].
- Proposed Evaluation and Test SW-CMM Key Process Areas (SW-CMM KPA) [6].

Weatherill concluded there were only two testing maturity models that were useable in their current format: SW-TMM and TPI. I read his article with extreme interest since I had already been searching for a testing maturity model to help my clients improve their testing processes. I judged the acceptability of a testing maturity model on the following:

- The ease of understanding and use.
- Allowing organizations to perform their own assessments.

"An assessment of the testing process using a testing maturity model ... will also highlight the variances between the imagined level and the actual level."

- The ability to provide a baseline of the current testing function and a road map for improvement.
- The capability of being used for telecommunications, Web-based, and information technology testing applications.
- The ability to be used in conjunction with SW-CMM.

I had already been researching the SW-TMM and, after reading Weatherill's article, I decided to further research the TPI. I came back to the SW-TMM as the best fit for my requirements.

Dr. Ilene Burnstein of the Illinois Institute of Technology and her associates designed the SW-TMM to be a companion to SW-CMM. Since 1996, she and her associates have published several articles in professional magazines, including the following:

- "Developing a Testing Maturity Model: Part I" [7].
- "Developing a Testing Maturity Model Part II" [8].
- "A Model to Assess Testing Process Maturity" [9].

The major weakness with SW-TMM is that there is no single book on the subject. I have corresponded with Burnstein, and the institute plans to release a book on the SW-TMM in 2002.

Why Should I Assess My Testing Maturity?

One of the biggest problems I have encountered while working with clients on their testing process is, that many times, there is no consistency within their organization as to the health and professionalism of the testing process. If you were to ask individuals at various levels of the organization their opinion of the current status of the software testing process, you will be surprised at the diverse answers you get. The answers given range from, "We have an excellent process in place, and don't need to change it," to "We have a horrible testing process. We should scrap it and start all over." The true level is usually somewhere in between these two extremes.

An assessment of the testing process using a testing maturity model will not only document the current level, but will also highlight the variances between the imagined level and the actual level. Only when the current status is known can significant improvements be made. Using the SW-TMM will not only help document the current level, but will also provide a road map for making the necessary process improvements.

What Are the Five SW-TMM Levels?

As previously stated, one significant advantage of the SW-TMM is its compatibility with the SW-CMM. It contains a set of five maturity levels, like the SW-CMM, with goals and sub-goals at each level that can be used as building blocks for improvement.

Level No. 1 is where most organiza-

℠ Testing Maturity Model and TMM are service marks of the Illinois Institute of Technology.

SW-TMM Goals Level 2 *Phase Definition*	SW-CMM v. 1.1 Key Process Areas Level 2 *Repeatable*
• Develop Testing & Debugging Goals	• Requirements Management
• Initiate Test Planning Process	• Software Project Planning
• Institutionalize Basic Testing Techniques & Methods	• Software Project Tracking and Oversight
	• Software Subcontract Management
	• Software Quality Assurance
	• Software Configuration Management

Table 1: *A Level 2 Comparison*

SW-TMM Goals Level 3 *Integration*	SW-CMM v. 1.1 Key Process Areas Level 3 *Defined*
• Establish a Testing Organization	• Organization Process Focus
• Integrate Testing into the Life Cycle	• Organization Process Definition
• Establish a Technical Training Program	• Training Program
• Control and Monitor the Testing Process	• Integrated Software Management
	• Software Product Engineering
	• Intergroup Coordination
	• Peer Reviews

Table 2: *A Level 3 Comparison*

tions start. Testing is a chaotic process. It is ill defined and not distinguished from debugging. The tests are developed ad hoc after coding is complete. The objective of testing at this level is to show that the system and software works. It usually lacks a trained professional testing staff and testing tools.

Most organizations will quickly recognize the need to develop a more organized and professional testing process. Many do not realize there is a structured process, like SW-TMM, available to make these improvements. Instead of using a structured process, they try to implement random improvement measures without a clear documented plan of approach.

When a testing process reaches Level No. 2, it identifies testing as a separate function from debugging. This is also the level where testing becomes a defined phase following coding. When an organization reaches this level, their testing goal is to show that the system and software meets specifications. They have standardized their process to the point where basic testing techniques and methods are in place. Table 1 shows a comparison of the SW-TMM goals and the SW-CMM key process areas (KPAs) at this level.

By the time the testing program reaches Level No. 3, the testing is integrated into the entire life cycle. The test objectives are now based on the system requirements. A formal testing organization is in existence. It establishes formal testing technical training, controls and monitors the testing process, and begins to consider using automated test tools. One of the major milestones reached at this level is that management recognizes testing as a professional activity. Table 2 shows a comparison at this level of the SW-TMM goals and SW-CMM KPAs.

This is the first level where testing is mentioned in activities five, six, seven, and nine under KPA Software Product Engineering. An organization should not wait until they arrive at this level to assess their testing processes for improvement. It is best to incorporate SW-TMM as a companion tool in the software process assessment (SPA) performed at Level 1 so that improvement can begin at Level 2. If an organization does decide to wait until reaching this level, two things may occur:

1. The costs associated with the improvements can significantly increase because of the sheer volume of necessary improvements.
2. Because of the magnitude of the process improvements and the time constraints, they may never get accomplished.

At Level No. 4, the testing is a measured and quantified process. The development products are now also tested for quality attributes such as reliability, usability, and maintainability. The test cases are collected and recorded in a test database for reuse and regression testing. Defects found during testing are now logged, given a severity level, and are assigned a priority for correction. Table 3 shows a comparison of the SW-TMM goals and SW-CMM KPAs at this level.

When an organization reaches the highest maturity level, CMM Level 5, the testing is institutionalized within the organization. It is well defined and managed, and testing costs and effectiveness are monitored. At this level, automated tools are a primary part of the testing process and there is an established procedure for selecting and evaluating testing tools. Table 4 shows a comparison of the SW-TMM goals and SW-CMM KPAs at this level. It should be noted that they are both concerned with defect prevention at this level.

Table 3: *A Level 4 Comparison*

SW-TMM Goals Level 4 *Management and Measurement*	SW-CMM v. 1.1 Key Process Areas Level 4 *Managed*
• Establish an Organization-Wide Review Program	• Quantitative Process Management
• Establish a Test Measurement Program	• Software Quality Management

Table 4: *A Level 5 Comparison*

SW-TMM Goals Level 5 *Optimization Defect Prevention & Quality Control*	SW-CMM v. 1.1 Key Process Areas Level 5 *Optimizing*
• Application of Process Data for Defect Prevention	• Defect Prevention
• Quality Control	• Technology Change Management
• Test Process Optimization	• Process Change Management

Why Do I Need to Use SW-TMM?

Now that we have discussed the five levels of the SW-TMM, the question on everyone's mind is: "The theory is nice, but why do I need to use SW-TMM?" If your organization is using the SW-CMM, the answer is obvious: SW-TMM is an excellent companion tool. The next question will probably be, "What makes SW-TMM an excellent companion tool?" The answer is that SW-TMM was designed to be a companion to SW-CMM. SW-TMM performs the following:

- Provides a methodology to baseline the current test process maturity.
- Is designed to guide organizations in selecting process improvement strategies and identifying the few issues most critical to software test process maturity.
- Is designed as an evolutionary path that increases an organization's software testing process maturity in stages.
- Provides a road map for continuous test process improvement.
- Provides a method for measuring progress.
- Helps an organization satisfy activities five, six, seven, and nine in Level 3 under KPA Software Product Engineering.

The first five bullets are virtually identical to the SW-CMM objectives. Organizations have to satisfy the activities listed in the last bullet in order to achieve Level 3. The SW-TMM provides a process, which can be incorporated into their SW-

CMM structure, for accomplishing those activities. Furthermore, the figures above illustrate that the goals of the SW-TMM complement the KPAs of the SW-CMM at every level. It is also easy to understand and use. Thus, I believe that SW-TMM fulfills the design objective of being an excellent companion to SW-CMM.

If an organization is not using the SW-CMM, they can still use the SW-TMM as a stand-alone tool to help improve their test processes. We know the following about this versatile tool:

- We know the basic elements of the SW-TMM.
- We know that the SW-TMM was designed to be a companion to the SW-CMM.
- We know that many organizations do not know their true testing maturity level.
- We know that performing an assessment using the SW-TMM will baseline an organization's current testing maturity level.
- The SW-TMM will help an organization map incremental improvements.

The next step in the process is to determine your organization's current testing maturity level. The only way to document the true level of testing maturity is to perform an assessment. If your organization is using the SW-CMM, then the SW-TMM can easily be incorporated into the SPA. It becomes just another assessment tool. As previously stated, the SW-TMM is not a tool to be used *in addition to*, but it is designed to be used *in conjunction with* the SW-CMM.

If your organization is not using the SW-CMM, then management will not approve making improvements to the testing process unless you can prove to them that it truly needs improving. They will not spend the money just because the SW-TMM is a *really neat* tool. Here are a few selling points that might be used with management: 1) the SW-TMM will provide an unbiased assessment of the current testing process, 2) the SW-TMM will provide a road map for incremental improvements, and 3) as the testing process moves up the maturity levels, there are usually some cost savings. Do not push for a commitment to implement the SW-TMM. Instead, consider it a win if you can get management approval to perform an independent testing process assessment using the SW-TMM.

Can Our Organization Perform Our Own Maturity Assessment?

The answer is "Yes." (Remember that one of my criteria when I evaluated the various maturity models was "allowing organizations to perform their own assessments.") In fact, an organization must perform their own assessment to feel ownership and have confidence in the results.

It is usually advantageous to hire a consultant to lead you through the process the first time. A consultant has performed the process before and can help reduce the learning curve. They also offer an unbiased perspective when analyzing the results and developing an action plan. The following suggested process works well either incorporated into the SW-CMM SPA or used as a stand-alone assessment.

How Do I Perform the SW-TMM Assessment?

The logical first step in assessing testing maturity is the *assessment preparation*. Now is the time to choose a team leader and team members. This team should develop the assessment plan and prepare the tools they will be using. Do not limit the assessment to just the testing organization. Include individuals – from senior management to the non-technical developer – from across the entire organization. These individuals should be either directly or indirectly involved with the testing process. You want to sample as many different and varied opinions as possible.

One of the evaluation tools that will be used is a questionnaire. I have modified the questionnaire for the TMM developed by the Illinois Institute of Technology that I make available to my clients. The questionnaire provides structure and consistency to the process and makes it easier to identify the current level of maturity. During this phase, it is essential to conduct all training and management briefings. The training and briefings educate everyone on the objectives and evaluation process to be used. Periodic management status briefings should also be scheduled at this time.

Once all of the preparations have been completed, it is time to *conduct the assessment*. The first step is to collect and record information. Here are some of the methods that can be used:

- Request the organization being evaluated to prepare a presentation and briefing for the team. This gives them an opportunity to present any information they feel is important from their perspective. It also demonstrates to them that they are an integral part of the assessment.
- Conduct interviews with all individuals on the assessment list. During the interview, the team members should complete the questionnaire.
- Review and photocopy all testing documentation and procedures to determine the actual testing process currently being used.

One of the most important assessment activities is to *document the findings*. By compiling all of the information collected by the team, they should be able to do the following:

- Document the organizations' current testing process based on the records and documentation review.
- Compile and summarize the questionnaire data using either a spread sheet or database program.
- Document the interview information. It is best if more than one person has conducted the interview. The interviewers will compare notes and document all agreed information.

While the documentation process is under way, it always becomes apparent that the team has missed some essential information or needs clarification of information. Now is the time to secure that information or clarification.

The assessment report should include a section containing the *analysis of findings*. The analysis should document the current maturity level and any areas of disagreement highlighted during the evaluation. It should also identify areas for improvement and a prioritized list of recommended improvement goals. The recommendations should include anticipated benefits resulting from implementation.

Usually the team will discover testing processes that are excellent, but are not utilized throughout the entire organization. I like to call these the best-of-breed processes. The team should include as many of the *best-of-breed* processes as possible in the improvement plan. There are several reasons for this:

- There will be better acceptance of the improvement plan if the team recommends building on current processes.
- It will accelerate the implementation and improvement process.
- People enjoy the feeling of pride that accompanies having one of their processes adopted organization-wide.

It should be emphasized that it is important for an organization not to try to progress from Level 1 to Level 5 in one giant step. That will result in almost certain failure. The recommendations should be a road map of how to reach only the next level of maturity.

The assessment team should *develop an action plan* for implementing the recommendations. The plan should describe

each specific action, the resource requirements, and a recommended schedule for implementation. A cost/benefit analysis is considered helpful supporting documentation. The action plan should be an integral part of the final report.

While a written final report is essential, a management briefing of the findings and recommendations should also be scheduled. It is usually easier to secure management approval of the recommendations after a management briefing. The written report should be provided as supporting documentation for the briefing.

After securing management approval to implement the improvement plan, it is time to *implement the improvements*. It is usually best, if possible, to implement the improvements either in a pilot project or in phases. This allows the organization to track progress and achievements prior to expanding organization wide.

Implementing in a limited application also makes it easier to fine-tune the new process prior to expanded implementation. Since the SW-TMM assessment process is repeatable, improvements can easily be tracked by repeating the assessment six months to a year after implementation.

Summary

The SW-TMM was designed as a companion to the SW-CMM to evaluate an organization's current testing maturity and to plan test process improvements. It accomplishes that goal. To recap, the use of the SW-TMM will provide the following:

- Baseline the current testing process level of maturity.
- Identify areas that can be improved.
- Identify best-of-breed testing processes that can be adopted organization-wide.
- Provide a road map for implementing the improvements.
- Provide a method for measuring improvement results.
- Provide a companion tool to be used in conjunction with the SW-CMM.

Clients who are using the SW-CMM that I have exposed to the SW-TMM can immediately recognize that it is an excellent companion tool. It can be easily incorporated into their SPA, thus helping them map the test process improvements necessary to reach the next level of maturity. Organizations not using the SW-CMM will also find the SW-TMM an excellent tool to realize their goal of improving their testing process.◆

References

1. Weatherill, Terry. "In the Testing Maturity Model Maze." Journal of Software Testing Professionals Mar. 2001: 8-13.
2. Gelperin, David, and Hayashi Gelperin. "How to Support Better Software Testing." Application Trends May 1996.
3. Kooman, Tim, Martin Pol, Henk W. Broeders, and Hans Voorthuyzen. Test Process Improvement. Addison-Wesley, May 1999.
4. <www.evolutif.co.uk>.
5. Software Futures Ltd.
6. Weatherill, Terry. "In the Testing Maturity Model Maze." Journal of Software Testing Professionals Mar. 2001. <www.softdim.comliist/journal.htm>.
7. Burnstein, Ilene, Taratip Suwannasart, and C.R. Carlson. "Developing a Testing Maturity Model: Part I." CrossTalk Aug. 1996.
8. Burnstein, Ilene, Taratip Suwannasart, and C.R. Carlson. "Developing a Testing Maturity Model: Part II." CrossTalk Sept. 1996.
9. Burnstein, Ilene, Ariya Homyen, Robert Grom, and C.R. Carlson. "A Model to Assess Testing Process Maturity." CrossTalk Nov. 1998.

About the Author

Thomas C. Staab owns an independent consulting firm, Wind Ridge International, which helps clients improve their software quality assurance and testing processes. He has more than 35 years experience in information technology and quality assurance. Staab has a master's of science degree in quality systems and is listed in the "International Who's Who of Information Technology." He has currently published more than 25 articles and presented more than 50 speeches and tutorials at regional, national, and world conferences.

Wind Ridge International
11321 E. Folsom Point Lane
Franktown, CO 80116-9105
Phone: (303) 660-3451
Fax: (303) 660-2057
E-mail: tcstaab@windridgeinternational.com

Reality Configuration Management

Donald E. Casavecchia
ACS Defense, Inc.

You are not alone if you have found that in your job as configuration management (CM) lead, you are given less than optimal support for your task, or are asked to scale back your CM goals. This author faced these dilemmas in his CM position at a small systems development facility. Here is how he adjusted his CM practices based on facility resources and management's commitment to CM.

Do you work for a small systems development facility? Does management profess a desire to implement the Software Engineering Institute's (SEI) Capability Maturity Model® [1] (CMM®) by next fiscal year? Are you the one they hired to miraculously transform their hobby shop into the lean mean systems generating machine they envision? Are you getting something less than the 100 percent support you were originally promised?

Three years ago, I joined a small systems development facility as configuration management (CM) lead, a newly created position, and was initially tasked with getting their software under control. This is a government facility with engineering contractors supplying the labor, and government engineers filling management positions as technical advisors.

Once past the security clearance barrier, I determined the facility was consistently in the process of developing some 20 separate projects simultaneously, each with six or less project members, with start to finish schedules ranging from three months to two years. As soon as one project ships its deliverables, another proposal is turned into real work, and a new project is kicked off. Post-delivery system support ranged from no support to the full operations and maintenance (O&M) regiment. Project team members are a mix of seasoned engineers and technicians that build complex systems entirely behind closed doors with project-obtained resources.

Prior to my arrival, senior management provided formal CMM training for every employee. One year later, project managers and technical advisors were required to attend repeat CMM training sessions. The facility chief and deputy appeared to want repeatable process-oriented systems development for their facility. They failed, however, to set forth the policy and direction to accomplish it. Their desire to step up a level from producing ill-managed prototypes to cost- and schedule-driven first articles with detailed build-to documentation was not being realized.

They held an *all-hands* on-site briefing to emphasize improvement goals. Unfortunately, they directed their frustrations down their own organization rather than coordinate across the customer base for better quality assurance requirements and adherence to stricter standards. Without customer requirements for repeatable processes with meaningful milestone reviews, the underlying work ethic remained "do only what it takes to get the job done."

> *"Instead of several CM tools from competing vendors, one was selected as the center's standard, and training became an across-center effort instead of each project sending their engineers for vendor-supplied training."*

By interacting with project members, I was able to identify the following reoccurring CM deficiencies:

- Vague, often undocumented requirements.
- *Rough-order-of-magnitude* proposals with cost and schedule estimates usually provided before requirements were firm.
- Follow-up project plans that failed to provide enough detail for project members to understand what it was they were building.
- No commitment to make project plans living documents.
- Chassis, cable, printed wiring board (PWB), mechanical, and schematic drawings too loosely controlled, with far too many redline variations that contributed to *best guess* build-to documents.
- Lack of rigid inspection checkpoints on drawings and PWB build-ups.
- Minimal software design documentation and few written unit test plans.
- No agreed-upon milestone identified for starting formal change management.

After several sessions with the lead system engineer during several months, I concluded that our small systems development facility, with less than 70 contractor and 15 government employees, could not dedicate the resources to establish a systems engineering workgroup and charter it with developing and implementing center policies, processes, and procedures. I witnessed our lead engineer receiving even less upper management support than I received. Eventually, he was dismissed from the program (and not replaced). It was apparent that if I wanted to improve CM practices, it would require a *grassroots* approach.

My first three months were spent developing a *makefile* [2] build system and converting a project's CM system from a homegrown source code control system (*sccs*)-based system [3] to one based on a commercial off-the-shelf (COTS) CM product. This quickly established me as a hands-on team player and gained me the support of key engineers and managers.

With one *fire* extinguished, I still had 19 other projects in need of CM improvements. With nobody yelling fire, I persuaded management to let me design and establish a local area network (LAN) to install and maintain a common set of engineering tools to be used across projects. Instead of each project purchasing, installing, and maintaining their own development environment on stand-alone workstations or makeshift workgroups, we pooled selected project products, switched from node-locked to floating licenses where possible, and established a

Categorized	Project Type	Risk
Class 1	Concept Design	High
Class 2	Development Design	Moderate to High
Class 3	Integration Design	Moderate
Class 4	Application Enhancement	Low to Moderate
Class 5	Application Maintenance	Low
Class 6	Production	Low

Table 1: *Classes of Projects Defined*

development infrastructure.

Previously isolated workspace offices now received LAN drops, meaning workstations could utilize the LAN to allow project developers to access infrastructure products. By providing a common infrastructure product base for Windows and Unix platforms, management could budget and coordinate training targeting infrastructure products and set policy and procedures for project teams to follow. Instead of several CM tools from competing vendors, one was selected as the center's standard, and training became an across-center effort instead of each project sending their engineers for vendor-supplied training.

After using this evolving, controlled infrastructure for three years, our center defined six classes of projects (Table 1).

We found that we often pursue a concept design project that later spawns separately funded integration design and/or production projects. Fielded systems often call for incremental advancement of a design (Class 2) or major enhancement (Class 4) projects. Each new request for a proposal is now categorized as Class 1 through 6, and each class carries predefined level-of-effort disciplines like CM, quality assurance, and documentation support.

Most of our Class 1 and 2 projects fall under the general descriptions of *proof of concept*, *investigate leading edge technology*, *rapid development of a prototype*, *conduct a trade study on xyz technology*, etc. These projects are usually short-scheduled with limited funding. Often, they are meant to only convince the customer that we could exploit the technology and deliver a system. Because we often build a *working model* or *prototype* and often write white papers as part of these project executions, capturing the more important parts of the project is all CM is able to achieve; often, we receive a media with the soft copy deliverables.

Sometime later (weeks, months) we may get tasked with revisiting the earlier effort and building a first article to demo. The follow-up task is a new project, separately funded with additional requirements. Since we have proved the concept, it is now less a risk and more an existing technology Class 3 or 4 project.

The following four sections in this article depict CM methods available to our project managers to satisfy CM requirements for the six classes of potential projects with which this center is involved. When asked to quickly (less than 90 days) produce a narrowly defined system (Class 2 project), the Archival Method is appropriate. The Archival Method is selected for requests of additional copies of a system we designed and built 18 months ago as an integration design project using the Open Repository CM Method (the additional copies would be categorized as a Class 6 project).

The Open Repository Method is usually appropriate for a concept design (Class 1) or development design (Class 2) project where schedule is usually longer than three months, and deliverables are often prototypes or working models.

The Focused Repository Method is always appropriate for our *bread-and-butter* systems that have proven themselves and when a hardware and/or software enhancement (Class 4 project) is requested. The Focused Repository Method is usually selected when problems are reported (Class 5) on fielded systems for which our center is on the hook for lifecycle support.

With the bulk of our delivered systems living short life cycles (mostly due to technology advancements), overCMing can be a real cost and schedule issue. If/when a fielded system (Class 3 project) exceeds expectations and takes on a long life cycle (greater than four years) with requests for additional copies with expanded functionality, we sometimes have to resurrect an Archival Method repository and bring it up to Focused Repository levels of CM resource commitment.

We have yet to achieve a system that provides enough metrics to seriously examine and tune our CM processes (Optimized Repository). I look forward to that day. The "CM Discipline Progression" depicts our least restrictive to our most restrictive CM method. I would love to report that every time our center has selected minimal CM (Archival Method) for a project, we have not regretted it. Likewise, we have gone all the way with the Focused Repository Method only to watch our system sit on the shelf with no takers.

Archival Method

The Archival Method (characterized as a *capture* technique) is selected when minimal CM is appropriate. The program manager (PM) specifies the schedule milestone(s) at which the baseline will be archived and identifies the set of system components for capture. Minimum CM occurs when the selected milestone is System Acceptance Test (SAT) and an O&M phase is not specified. When multiple milestones are designated, or an O&M phase is required, all soft copy files associated with the milestone should be placed into a project repository and *labeled* with the milestone acronym. CM technicians work closely with project members to catalog system components, down to lowest replaceable units (LRUs), comprising the project at the specified milestone.

CM Requirements

The PM is responsible for identifying the set of system components to be archived. Software system components, comprised of source files (no intermediate or build product files), are isolated from project work areas (preferably placed on a transfer media) after the following is verified:

- Builds cleanly, without errors.
- Successfully executes.

Each software system component's build and execution (run-time) environment must be documented to ensure its reproducibility. The following are the minimum details to include:

- Development and target platform nomenclature (if appropriate), including identification of any special boards, cables, peripherals, and drivers.
- Operating system version and list of patches.
- COTS and/or government off-the-shelf (GOTS) version, installation order, feature selections, configuration files, patches, and integration code.
- Compilers, linkers, and loaders, including their version and switch settings.
- Environment variables and their settings.
- Dependencies on any third party libraries, identify source and version, and use restrictions.
- Actual license certificates, keys (dongles), and maintenance agreements.
- Test tools, either internally developed

or commercial.

Each soft copy document turned over to CM should be saved in a format that turns off any tool propriety revision display feature[4]. This is especially important for drawings from computer-aided design COTS packages.

Hardware turned over to CM (usually for transfer to an external O&M facility) will be appropriately identified and classified. Bill of materials (BOM) must be detailed down to line replacement units (LRU). Any special handling or environmental stowage requirements must be made known at the time of turnover.

Strengths

- Simplifies project member's work environment.
- Allows staffing the project with less experienced workers.
- Minimizes impact to project members.
- Reduces training needs.
- May result in shorter system development timelines.

Weaknesses

- Places most of the responsibility for executing CM onto CM technicians who least understand the project's organization, goals, and deliverables.
- With respect to version control, this method merely captures a snapshot set of system components corresponding to the designated milestone. Component versions created between archived snapshots are lost.
- With respect to change management, when issues/problems are not documented or processed via a review/approval process, management also has no insight as to the number of problems fixed (product quality) or the way that problems are fixed (design quality).
- With respect to configuration control, this method frequently requires an inordinate effort from CM to configure the baseline, i.e., understand the project components hierarchy (software file system restructuring is perhaps the worst case) and identify and apply a meaningful software labeling scheme.
- With respect to status accounting, little or no metrics are available or collected. It is difficult to associate between changes to components and the driving requirement, e.g., no way to track revision three to system component X with the corresponding issue/problem report.
- With respect to auditing, no formal baseline exists until a *capture* milestone is executed. Customer insight to completed work is not verifiable. Change implementation is invisible.

Open Repository Method

The Open Repository Method (characterized as a *contribution* technique) is selected when management desires closer control and progress review capabilities. Project members are tasked with routinely submitting soft copy versions of their work into a project repository. More dynamic and comprehensive than the Archival Method, the Open Repository Method ensures that aggregate changes to a component are contributed to the repository as an identifiable version. Typically, an *add to* or *check-in/check-out* interactive exchange is employed to mature the repository from project start-up through all phases of the project's life cycle. Management review of a project is greatly simplified when every project member is conscientiously contributing to and/or entering changes into the repository at predefined milestones.

CM Requirements

It is necessary to comply with each Archival Method requirement in addition to the following:

- Pre-coordination and agreement between CM and the project of a file system structure to accommodate all known system components and their interfaces.
- As much advanced notice as possible on project selected development tools to allow for interoperability evaluation to determine the best way to save and stow soft copy files from these tools into the repository.
- A mandatory repository check-in comment that appropriately describes the aggregate of changes to a component since last check-in.

Strengths

- Anyone with permission to access the repository can monitor when project components enter the repository as change sets at designated milestones.
- Work is centralized to a single file system simplifying backup procedures.
- Facilitates project member communications because everyone knows where to look for project items.
- Provides integration between development tools (like Microsoft's Word and Visual C++) and the repository for direct check-in/check-out processing right from the tool they are using; with some tools, compare and merge capabilities can exist.
- Increases the opportunity for *common* software and software reuse.
- Provides management with meaningful metrics on project and component size and complexity. The number of versions for an item may convey the level of development difficulty or number of problems overcome.
- Reduces the data entry workload on CM by distributing the repository entry responsibility to each project member.

Weaknesses

- With respect to version control, although the open repository method may produce file versioning, without an organizational policy that mandates following a change-management procedure for each file update, discrete version control is not being exercised.
- With respect to change management, although the open repository method allows a comment to be entered when a repository file is added, modified, or replaced, without a formal issue management process with board adjudication, change management is not being exercised. Repository changes are not subject to formal review.
- With respect to configuration control, without formal change management, controlling a project's configuration by defining labels that correspond to schedule milestones and manually apply them is about the best you can achieve. The open repository method does not achieve verifiable baseline advancement. An opportunity exists for unsolicited enhancements.
- With respect to status accounting, version metrics and associated comments are available and can be reported, but correlation of problems fixed to specific files changed is still missing.
- With respect to auditing, it provides both management and customer the opportunity to review soft copy files (including schedule updates) in the repository, but fails to provide issue-management metrics (action item, issue, problem report, engineering change proposal, engineering change notice, revision-level change, etc.).

Focused Repository Method

The Focused Repository Method (characterized as a *directed* technique) is selected when management has CM policies/procedures institutionalized within their organization and the intent is for full control of workflow processes for the project[5]. Project members are indoctrinated on CM policies, issue documenting, and

change management procedures. The PM is indoctrinated on management review policies and the various metric reports available from status accounting.

The Focused Repository Method is a disciplined process-oriented approach to achieving CM during project execution. All project members have access to the issue-tracking tool and all issues are documented when they become known. The timely review and disposition of every issue conforms to the organization's change management process. For our organization, two Configuration Control Boards (CCBs) handle the disposition of issues. A project-level CCB handles all issues that do *not* affect schedule, cost, or design. A program-level CCB dispositions all schedule-, cost-, and design-related issues. Management overview of project execution is near real-time because issues are immediately surfaced and dealt with.

CM Requirements

It is necessary to comply with each Archival and Open Repository Method requirement in addition to the following:

- Project plans must identify major system components and supply support documents that fully describe:
  - Hardware components detailed to subassemblies, LRUs, with drawings, board layouts, and accurate BOM and formal build-to documentation.
  - Drawings and layouts comply with IPC-A[6] revision standards.
  - Firmware, vendor supplied or custom developed, maintenance plan.
  - Bundled COTS, GOTS, version, license and distribution agreements.
  - Software components, build order, build mechanism, build environment, run-time environment, release strategy, version description, and maintenance plan.
- Project members must attend CM training sessions covering tools, processes, and procedures.
- Project engineers responsible for tool selection must coordinate with CM on tool integration and upgrade tasks.

Strengths

These include all the strengths listed under the Open Repository method plus the following:

- Verifiable change management.
- All repository changes are subject to formal review.
- All changes are captured.
- Project issues are documented, adjudicated, and dispositioned resulting in traceable system component changes within the maturing baseline.
- Provides for accurate file compare capability, a single change set that directly correlates with a single issue.
- Management has full insight into the number of problems fixed (product quality).
- Management can review exactly how an issue was fixed.
- Reduces CM technician's involvement with repository input.

Weaknesses

- Requires organizational policy, procedure, and training programs, each subject to a continuous improvement effort.
- Customer buy-in, including compiling adequate cost and schedule metric briefings to convince customers that implementing effective CM gets them better products at cheaper prices within shorter development cycles.

"A small systems development facility can achieve limited CM proficiency by selectively implementing CM disciplines across their business lines."

Optimized Repository Method

The Optimized Repository Method (characterized as a *tuning* technique) is selected for a project by management only after an acceptable number of projects using the Focused Repository Method have been completed and thoroughly evaluated. Isolated areas with weak processes and procedures, insufficient metric collectors, inadequate change-tracking information, forms and route slip inadequacies, high quality control failure areas, and high percentage test failure components are targeted for having their workflow processes fortified. Fortifications include the following: more stringent reviews, tighter version management, better testing (additional regression tests), additional required fields for capturing metric data, and a higher degree of system decomposition. Management targets a specific project for observation and evaluation using the improved CM practices.

CM Requirements

It is necessary to comply with each Archival, Open Repository, and Focused Repository Method requirement in addition to the following:

- Project members must attend CM training sessions covering tool, process, and procedure enhancements or replacements.
- Project members must comply with entering additional form and route slip inputs.
- Project members must supply greater detail to required comment fields when assigned issue and change resolution tasks.

Strengths

These include all strengths listed under the Open and Focused Repository Methods plus the following:

- The project is first to try out new tools, processes, and procedures.
- Increased metric data usually results in more accurate cost and schedule reporting.
- Companies executing disciplined systems development have an advantage when it comes to attracting good engineers.
- Companies executing disciplined systems development have the advantage over undisciplined companies that cannot bid (SEI/CMM competition).

Weaknesses

- The project is first to try out new tools, processes, and procedures.
- May result in a longer system development timeline.
- Tendency for higher project cost associated with implementing process changes.

Conclusion

A small systems development facility can achieve limited CM proficiency by selectively implementing CM disciplines across their business lines. As management realizes benefits from the relatively small resource investments associated with the Archival Method, a natural progression to Open and Focused Repository Methods becomes almost automatic as customers communicate their desire or need for system advancements.

Conversely, when the deliverable is a working model that proves a concept, or a design, cost, and schedule are the paramount requirements, Archival Method processes may be just the right amount of

CM. Government contracts awarded to the *low bidder* demand cost effective proposals with the basis of estimates receiving careful scrutiny. Minimizing CM often becomes a target for cost savings for small developing facilities facing the reality of having to present a winning proposal.◆

Notes

1. Substitute ISO9001, Capability Maturity Model® Integration℠, or Malcolm Baldrige Award, as applicable.
2. Makefile: A manually generated, specially formatted input file to the *make* utility, which contains information about what files to build and how to build them. The *make* utility streamlines the process of generating and maintaining object files and executable programs. It helps to compile programs consistently, and eliminates unnecessary recompilation of modules that are unaffected by source code changes.
3. A source code control system (SCCS) allows you to control write access to source files and to monitor changes made to those files. Allows only one user at a time to update a file, and records all changes in a history file. SCCS is a bundled Unix utility.
4. Our center's CM tool is integrated with our desktop office package to automatically display two versions of a non-binary file in revision mode.
5. Our center selected Rational's ClearCase product as its standard for creating and maintaining individual project repositories because ClearCase allows engineers, technicians, and managers to work directly in the project repository.
6. IPC: From 1957-1999, IPC stood first for *Institute for Printed Circuits*; later it became *Institute of Interconnecting and Packaging Electronic Circuits*. In 1999, IPC changed its name to just plain IPC, which has its offices at 2215 Sanders Road, Northbrook, IL 60062-6135. The IPC provides the following information:
   - Standards to facilitate communications between suppliers and customers.
   - Guidelines with current industry positions on a wide range of subjects.
   - Research to solve industry problems.
   - Correlation of industry test methods.
   - New developments in interconnection technology.
   - A monthly publication called Relay.
   - Provides training (and certification) at U.S. and overseas sites.
   - Maintains a web site at <www.ipc.org>.

   For example: IPC-A-610C – Acceptability of Electronic Assemblies: This standard is a collection of visual quality acceptability requirements for electronic assemblies.

About the Author

Donald E. Casavecchia is director, configuration management/quality assurance (CM/QA) with the Warrenton Operations Strategic Planning Group of ACS Defense, Inc. He is a hands-on CM manager with more than 20 years experience with the Department of Defense and government projects, including assembly, BASIC, FORTRAN, C, and script programming, setting up and maintaining software development environments, and supporting embedded systems development. Casavecchia believes in implementing practical CM/QA solutions.

ACS Defense, Inc.
P.O. Box 700
Warrenton, VA 20188
Phone: (540) 349-3762
Fax: (540) 349-3517
E-mail: donc@wtc.gov

WEB SITES

DACS

www.dacs.dtic.mil/

The DACS is a Department of Defense Information Analysis Center. The DACS supports the development, testing, validation, and transitioning of software engineering technology to the defense community, industry, and academia. DACS' subject areas encompass the entire software life cycle and include software engineering methods, practices, tools, standards, and acquisition management. Also included are programming environments and language techniques such as Ada and Object Oriented Design, software failures, test methodologies, software quality metrics and measurements, software reliability, software safety, cost estimation and modeling, standards and guides for software development and maintenance, and software technology for research, development, and training. The DACS is a central distribution hub for software technology information sources.

Defense Acquisition Deskbook

www.web2.deskbook.osd.mil/5000Model.asp

The Defense Acquisition Deskbook is sponsored by the Office of the Under Secretary of Defense. It functions as a hub for the Department of Defense's (DoD) acquisition center, including the DoD 5000 Model, quick links, reference library, ask a professor, special interest items, education and training, and more.

Software Testing Institute

www.softwaretestinginstitute.com

The Software Testing Institute (STI) provides access to quality industry publications, research, and online services. STI offers a software testing discussion forum, the STI Resource guide, and privileged access to STI's industry surveys on salaries, industry trends, staffing and more.

CrossTalk

www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering, introduces its newly redesigned Web site. We have completely revised our look and links making it easier and faster for you to locate any issues, read back issues, review current monthly themes, search the Web site, download author guidelines, submit questions and feedback, and more. CrossTalk is an approved Department of Defense journal. Our mission is to encourage the engineering development of software in order to improve the reliability, maintainability, and responsiveness of our war fighting capability and to inform and educate readers on policy decisions and new software engineering technologies.

Software Program Managers Network

www.spmn.com

The Software Program Managers Network (SPMN) is sponsored by the deputy under secretary of defense for the science and Technology, Software Intensive Systems Directorate. It seeks out proven industry and government software best practices and conveys them to managers of large-scale Department of Defense software-intensive acquisition programs. SPMN provides consulting, on-site program assessments, project risk assessments, software tools, guidebooks, and specialized hands-on training.

StickyMinds

www.stickyminds.com

StickyMinds.com is designed for software managers, testers, and quality assurance engineers to share information and is the Web site where they gather to discuss know-how. StickyMinds.com is all about gathering knowledge.

Document Diseases and Software Malpractice

Gregory T. Daich
Software Technology Support Center/SAIC

This article proposes some names for software documentation diseases based on human diseases that will surely invoke a feeling of alarm to motivate developers to plan immediate treatment to keep project costs and quality under control. To ignore and leave these documentation diseases untreated may one day be considered software malpractice. This article can help to identify serious documentation maladies early when treatment is possible to maintain a lean, healthy, and on-time project.

Disciplined document reviews are somewhat analogous to medical instrument sterilization. Today, we would not tolerate doctors using unsterilized equipment to conduct surgery. However, it was common practice, for example, during the Civil War to use a surgical instrument such as a saw without proper sanitation between limb amputations. This caused more soldiers to die of infections than from the initial wounds. Once scientists learned about bacteria and other microbes, the medical industry began requiring that equipment be sterilized to avoid infecting patients.

Today, we have plenty of experience data that shows disciplined document review practices are vital to delivering systems on-time and within budget [1, 2]. However, many software development organizations still have no problem using yesterday's practice of whipping together a set of requirements and delivering it to the developers and testers without an adequate review. They do not take the time nor use effective tools to look for document defects. Either they do not see any obvious bugs, or they must figure the bugs can be fixed later. Consequently, they do not seem to mind delivering a document with serious diseases, so to speak. However, it is a well-known fact that defects cost more to fix the longer they remain in the system documentation (and code) [3].

Yesterday's state-of-the-practice is often today's malpractice. Let me go on record as saying that organizations not implementing a disciplined document review program will one day be considered to be conducting software malpractice. Disciplined document reviews are that important!

Disciplined document reviews are also known as peer reviews, inspections, and structured walk-throughs. While there are some differences in definitions for these terms in various standards and guidelines [4] in the industry, the intent is to identify defects to determine if the document is ready for release to the next phase of development or for delivery to the customer. The author should prepare the document to the point that he or she believes that there are few, if any, defects remaining, and that it is considered ready for release to the next phase of development or delivery. Another major purpose for these reviews is to identify process improvement opportunities to avoid making the same mistakes on future efforts.

Human Disease Analogies

In any discipline, we provide names to characterize the topics, objects, and projects that we need to discuss. Naming a concept helps us gain an initial understanding of the issue. If naming document maladies after human diseases could help software engineers better understand and deal with the documentation problems confronting them, then I propose doing so. There are several document diseases that I have seen in my research at the Software Technology Support Center (STSC) and during my support of STSC clients. Since many of us are not readily conversant with human diseases, here are a few definitions for review [5]:

- **Rickets**. A deficiency disease resulting from a lack of vitamin D, marked by defective bone growth and occurring chiefly in children.
- **Sclerosis**. A thickening or hardening of a bodily part, especially from disease or excessive growth of tissue.
- **-itis**. Inflammation or disease of [a body part or parts].
- **Scurvy**. A disease resulting from a deficiency of vitamin C and marked by bleeding under the skin.
- **Glaucoma**. A disease of the eye marked by high intraocular pressure and partial or total vision loss.
- **Cancer**. Malignant neoplasms that manifest invasiveness and have a tendency to metastasize to new sites.

After this quick review, surely you can see that relating these names to document defects could invoke a feeling of alarm that would require immediate treatment to keep project costs and quality under control. The following sections discuss several document diseases to to treat immediately on your projects.

Requirements Rickets

Requirements rickets is a deficiency of important requirements resulting from a lack of exposing requirements to adequate analysis and review. For example, requirements state that a behavior shall occur given a specific condition. But they do not tell you what to do when the condition is not met. This is a common cause of requirements rickets: The symptom is missing requirements. Granted, an iterative or evolutionary development life cycle does not expect all requirements to be defined at the start. But when you define the requirements that you do know about, be sure they are adequately specified.

Another example of requirements rickets occurs when we state that we want a usable system and then do not provide any objective scale of measure for usability. We are missing vital information, without which we can neither build the desired system nor evaluate whether it has been successfully accomplished. Example scales of measure for usability include the average time to learn to use specific product features and the average time to perform specific product features by experienced personnel [1].

Source Code Sclerosis

Source code sclerosis is a hardening of the source code, making it very difficult to correct or upgrade without breaking some other existing capability. Many old legacy systems are built with breakable design architectures and poor coding practices (for example, inadequate comments, unstructured code, or poorly named variables). These systems require intensive care to revive them to prolong their lives many times. We often spend 70 percent or more of the entire life cycle doing maintenance on many of these systems [6].

Some of these systems with source code sclerosis need to be taken off life support. They need to be redesigned and rebuilt to cost-effectively meet operational requirements. With the support of automated static analysis tools, effective document reviews assures that maintainable code is developed.

Ambiguousitis

Ambiguousitis is a disease of ambiguity inherent in all natural languages and is as common as the common cold. Sometimes it provides some variety, mystery, or humor to fictional prose but it always causes literary dizziness and confusion in software documentation. People can legitimately interpret the same specifications in different ways. For example, consider the following statement: "If Transaction_A is received, and it is the end of the week, or it is the end of the month, produce Summary_A Report."

This text can be interpreted in at least two distinct and valid ways using the parentheses to clarify as follows:

- Interpretation 1: "If (Transaction_A is received, and it is the end of the week) or it is the end of the month, produce Summary_A Report."
- Interpretation 2: "If Transaction_A is received, and (it is the end of the week or it is the end of the month), produce Summary_A Report."

Making an assumption about what the original text means is dangerous to the health of the project. Interpretation 1 always produces Summary_A Report at the end of the month. It also produces Summary_A Report if it is the end of the week and Transaction_A has been received. Interpretation 2 produces Summary_A Report only if Transaction_A is received and either it is the end of the week or it is the end of the month. (Is that clear?) In other words, the results are different depending on the reader's interpretation. This disease has caused entire projects to die a slow and painful death, though the symptoms were often known very early and could have been diagnosed and treated.

Source Document Scurvy

Source document scurvy is a malicious disease resulting from a deficiency of source document references and is rampant in many project documents. Its effects are often tolerated as part of life when the quality of our projects could be greatly improved with the adoption of a few effective antidotes. If you obtain information from a source document, then provide a useful citation to that document such as a reference identifier with a page or section number (if it is not obvious where the information came from). It takes only a little more time and saves countless review and rework hours later.

Note that a reference section should always be provided in all documentation, and includes the titles, authors, dates, and version numbers of all sources. No document is an island. Some managers want the documents that they have to review to stand on their own. Thus the authors hide vital reference information by not listing all the applicable references. You can often find inconsistencies between documents when you check it under review against the source document.

Document Glaucoma

Document glaucoma is a disease caused by unclear or missing document and project objectives. It is a frequent illness in special reports, plans, or guidelines to support software development or maintenance efforts. When I am asked to review these types of documents, I often cannot find a statement of objectives. With no vision of where we are really headed in some of these documents, it is a wonder we make any progress at all.

Software-related documentation written by inexperienced authors often jumps right into the body of the document with no proper introduction. This may have something to do with the mindset of some developers that documentation is bad because it slows us down. That is like saying we do not have time to document our plans or our requirements. We just have time to build. "We'll document later," is a common comment I have heard from a variety of sources. This is a clear example of document glaucoma; unclear project objectives is its main symptom.

Content Cancers

Content cancers occur whenever an uncorrected, often malignant documentation problem is not corrected prior to delivery to the next phase of development. IBM studied how design defects propagate to multiple design defects, and how design defect propagates to multiple coding defects [6]. Again, documents written in early project phases often transmit these cancerous defects to follow-on phases because they were not diagnosed and the disease treated when authors had a chance – shortly after writing the document. This disease continues to grow maliciously to damage schedules and costs and ultimately kill projects. One particularly malignant strain of this disease manifests itself in "required" but trivial overkill documentation that does not support key project objectives.

Disease Cure and Prevention

Unlike some human diseases, these document diseases are completely diagnosable and curable using technologies available today. The technologies are not difficult but are a collection of common-sense activities that require training where effective practices can be experienced. However, effective document reviews require a significant process implementation effort following training. Many organizations do not make this vital investment.

Some of these common-sense activities include the following [1, 7]:

- Checking the document against objective criteria to determine readiness for review, and planning the review by a trained document review leader.
- Conducting a kick-off meeting to introduce the document to the reviewers and to answer questions.
- Checking the document against sources, checklists, standards and checking for ambiguities, incompleteness, inconsistencies, and missing sources.
- Meeting as a team of reviewers to report and check for additional defects.
- Conducting a process brainstorming meeting to begin root cause analysis shortly after the team review meeting.
- Correcting the document by the author or author-representative and addressing all issues and defects.
- Auditing the author's corrections.
- Verifying that objective document readiness criteria have been met by the review leader and recording metrics.

Once the review process is underway, it needs to be constantly monitored to assure that document review leaders are following the process. Without active management support and involvement, developers and support staff often slip back into archaic, skim-review malpractice (one quick gloss-over reading without checking sources and checklists). [1, 7, 8, 9, 10].

Poor document review practices persist in many organizations because there is no documented policy and process for conducting reviews; that is the way it has been done for years. Conducting haphazard ad hoc reviews allows critical project documentation to be transmitted unsterilized to subsequent phases of development.

We all know that "An ounce of prevention is worth a pound of cure." Effective document reviews also feed information back to developers to help improve the authoring process in the future.

Effective document reviews will not prevent all document diseases and will not guarantee success, but they go a long way toward maintaining healthy projects. They require up to 15 percent more time during the document authoring stages of development, but they save time overall on projects through significantly less rework and retesting [1]. It is like taking the time to eat well-balanced meals, participate in an effective exercise program, and get enough rest for your projects. You just cannot expect to live a healthy project lifestyle without effective and efficient document review practices.

With effective review practices, you can also expect more projects to reach full maturity. You will have fewer incidents of project euthanasia (cancellations) administered (which is probably the best idea for many suffering projects). You will also have more lean and healthy projects delivered on-time and on-budget [2].

Although I do not really expect any project to adopt my document disease naming convention, it brought to light some interesting issues about documentation maladies that are treatable today. One rule in conducting disciplined document reviews is to direct comments at the document, not the author. Naming document diseases this way may suggest that some authors are carriers of certain diseases. We do not really want to label authors this way. However, it may be fair to say that responsible managers who neglect implementing effective reviews of critical documentation may be the real carriers of the above document diseases.

The consequences of poor quality documentation can still result in preventable project fatalities just like individuals practicing poor health habits result in preventable fatalities. Again, to do anything less than implementing disciplined document reviews could eventually be software malpractice.◆

Acknowledgements

I would like to thank the following people for their comments regarding this article: Pam Bowers, Ross Collard, Rick Craig, David Dayton, Chelene Fortier, Tom Gilb, Paul Hewitt, Tony Henderson, Cem Kaner, Ed Kit, Bret Pettichord, Ron Radice, Johanna Rothman, Bob Stahl, Beth Starrett, Tracy Stauder, and Karl Wiegers.

References

1. Gilb, Tom, and Dorothy Graham. Software Inspection. Boston: Addison-Wesley, 1993.
2. Dion, Raymond. "Process Improvement and the Corporate Balance Sheet." CrossTalk Feb. 1994.
3. Boehm, Barry. Software Engineering Economics. Prentice Hall, 1981: 17.
4. Institute of Electrical and Electronics Engineers. "Standard for Software Review Processes". IEEE 1028-1997.
5. Webster's II New Riverside University Dictionary. The Riverside Publishing Company, 1984.
6. Collard, Ross, et. al. System Testing and Quality Assurance Techniques. Collard & Associates, 2001.
8. Ebenau, Robert G., and Susan H. Strauss. Software Inspection Process. McGraw Hill, 1993.
9. Radice, Ronald A. High Quality Low Cost Software Inspections. McGraw-Hill, 1993.
7. Daich, Gregory T. "Disciplined Document Reviews Course." Software Technology Support Center. Mar. 2002, Version 6.
10. Wiegers, Karl E. Peer Reviews in Software: A Practical Guide. Addison-Wesley, 2002.

About the Author

Gregory T. Daich is a senior software engineer with Science Applications International Corporation currently on contract with the Software Technology Support Center (STSC). He supports STSC's Software Quality and Test Group with more than 25 years of experience in developing and testing software. Daich has taught public and on-site seminars involving software testing, document reviews, and process improvement. He consults with government and commercial organizations on improving the effectiveness and efficiency of software quality practices. Daich has developed two Air Force training programs: Software-Oriented Test and Evaluation, and Disciplined Document Reviews. He has a master's degree in computer science from the University of Utah.

Software Technology Support Center
OO-ALC/MASEA
7278 4th St.
Bldg. 100
Hill AFB, UT 84056-5205
Phone: (801) 777-7172
E-mail: greg.daich@hill.af.mil

Defense Software Development in Evolution©

Capers Jones
Software Productivity Research, Inc.

The author and his colleagues have been measuring software quality and productivity rates since 1985. They classify the projects that they examine into six major groupings: information technology software, outsource software, commercial software, systems software, defense software, and end-user development software. Applications are placed in the defense software group if they followed U.S. military or Department of Defense (DoD) standards. Overall, defense software projects have ranked near the top in software quality. However, defense software projects have ranked last in terms of software productivity, mainly because DoD standards created a number of extra tasks for defense software that do not occur in the civilian sector. In addition, the volume of defense software specifications and other paper documents has been about three times larger than civilian norms. As the DoD moves toward adopting the best civilian practices and standards for software, it is possible to see some improvement in productivity while keeping software quality levels high.

The phrase *defense software* refers to software produced for a uniformed military service such as the Air Force, Army, Navy, Marines, or Coast Guard. The term can also include software produced for the Department of Defense (DoD), or the equivalent in other countries, by civilian companies such as Boeing, Lockheed Martin, Raytheon, and a host of others.

Uniformed military personnel produce some defense software. In the United States, however, civilian contractors develop the bulk of defense software. Oversight and project management roles are normally the responsibility of military program management officers.

The broad definition of defense software includes a number of subclasses such as software associated with weapons systems; with command, control, and communication systems (usually shortened to C3 or C cubed); with logistical applications; and also with software virtually identical to civilian counterparts such as payroll applications, benefits tracking applications, and the like. The main attribute that distinguishes defense software from other types of software is adherence to military or DoD standards.

DoD Software Industry Overview

Since about 1990, the U.S. defense contracting community has been undergoing some significant changes. Waves of mergers and acquisitions have led to a reduction in overall numbers of defense contractors. While the remaining contractors are growing in size, "downsizing" or elimination of redundant personnel also accompanied the mergers so the overall defense sector has not grown in terms of demographics for several years. However, in the aftermath of the Sept. 11 tragedy, new importance has been placed on the defense community so demographics may climb again in the future.

The United States is far and away the major producer and consumer of military and defense software in the world. The volume and sophistication of U.S. military software is actually a major factor of U.S. military capabilities. All those pictures of cruise missiles and smart bombs that filled television news during the Gulf War and the Afghanistan action have an invisible background: It is the software and computers onboard that make such weapons possible.

In addition, the NATO countries tend to use many weapons systems, communication systems, logistics systems, and other software systems produced in the United States. This means that the volume of U.S. defense and military software appears to be larger than the next five countries put together (Russia, China, Germany, United Kingdom, and France). Many other countries produce military and defense software for weapons and communications systems that they use or market, including Israel, Brazil, South and North Korea, India, Pakistan, Sweden, and Japan.

Differences Between Civilian and Defense Software Practices

To an outside observer, military software and hardware projects are noticeably different from civilian norms. The first noticeable difference is the procurement process itself. The bulk of military projects are acquired by means of competitive bids, with lowest cost as a primary consideration. The bidding process is quite formal and includes rather massive sets of deliverable items from the prospective contractors. Thus, responding to a military request for proposal can be an expensive proposition in its own right.

This form of acquisition by competitive bids also leads to another difference between civilian and military norms. Military procurement is often accompanied by litigation that challenges the successful bidder. Based on discussions with DoD officials, almost half of the initial contract awards are challenged by losing vendors. An entire body of military contract law, special courts, and arbitrators deal with these challenges. In contrast, less than 10 percent of civilian contracts go to litigation to challenge the winning bid.

As a result of frequent litigation challenging the initial contract awards, there is often a six- to 18-month delay in reaching a final decision on military software contracts and starting the work. This means that many large military software and hardware projects are under immediate schedule pressure. Since schedule pressure is one of the major root causes of software failures, some projects that are rushed tend to run late, have poor quality, or end up being canceled since they cannot meet operational requirements.

Another difference between military and civilian practice is readily apparent once the contract work begins. The relationship between the DoD and its contractors had tended to be somewhat adversarial. As a result, the oversight and control requirements of military projects have been more extensive and burdensome than civilian norms. This has had a direct and tangible impact on defense software productivity. On the other hand, the number of very large software projects successfully concluded in the defense domain appears to be better than civilian norms for the same sized applications. In other words, defense software may have more front-end litigation than civilian software, but fewer instances of litigation

for non-performance at the end.

Due to elaborate oversight requirements, the volume of planning and tracking paperwork required for a typical military software project has been about three times larger than for civilian software projects of the same size, based on our comparisons. Indeed, software requirements, software specifications, and almost all forms of text-based documents were several times larger for military projects than for equivalent civilian projects.

The large volume of paper documents is one of the main reasons why military software productivity rates lagged behind all other domains. About 400 English words were produced for every source code statement in the Ada95 programming language on typical military software projects in the 1980s and early 1990s. These words cost at least twice as much as the code itself. It is not unusual for large defense projects to accumulate roughly 50 percent of total costs in the area of producing and reviewing paper documents. This is far more than for any other kind of software.

While every software project needs requirements, specifications, plans, and deficiency reports, about half of the words created for military software projects seemed to be due to the very elaborate oversight and status reporting criteria associated with military contract work. Basically, some of the documents are produced to demonstrate contract compliance rather than to add technical content to the project itself.

Dealing with the DoD and the military services for business and contract purposes is so complex and specialized that companies actually doing significant amounts of military business usually have specialist military proposal and contract personnel who often are retired military officers. It is very difficult for amateurs to bid successfully on a military contract.

Being the world's largest producer and consumer of military software, the United States' software production methods are of global importance. In the United States in 1994, the Secretary of Defense William Perry issued a major policy statement [1] saying in effect that DoD standards no longer needed to be utilized. Instead, the armed services and the DoD were urged to adopt current civilian best practices.

Immediately, several task forces and study groups were created to explore leading civilian software practices. However, the military community has a conservative bent. Many military and DoD standards, such as MIL-STD-2167A or MIL-STD-498 [2], have continued to be the de facto standards of the military world, if for no other reason than because military contractors have used them for so long they are comfortable with the nomenclature and requirements.

Since civilian software fails, too, (witness the protracted delays associated with the luggage handling system of the Denver Airport [3]) another challenge for the defense community is to select practices from the civilian sector that truly do work, as opposed to practices that are merely fads. James Johnson and his colleagues at the Standish Group publish an annual report on software failures [4].

Further, some of the civilian standards, such as the ISO 9001-9004 quality standards, create document volumes that are just as large, or larger, than military standards such as MIL-STD-2167. Overall, selecting the *best* standards in either civilian or defense sectors is not necessarily an easy task.

"Organizations at or above CMM Level 3 are more likely to be successful on large systems [larger than 10,000 function points or 1,000,000 source code statements] than those at Levels 1 or 2."

The World Wide Web has many interesting sites dealing with the evolution of *military standards* toward civilian equivalents such as those published by well-known standards organizations, including the Institute of Electrical and Electronic Engineers (IEEE). Using a search engine with the key words "military standards" will bring up more than 20 relevant sites. One of the relevant documents is an interesting report on "Systems Engineering Standards and Models Compared" by Sarah Sheard and Dr. Jerome Lake. This document is available via the Software Productivity Consortium at <www.software.org/pub/externalpapers/98042.html>. On the whole, the best models for the military domain would be the large civilian systems software producers such as AT&T, IBM, etc.

The phrase *systems software* refers to software applications that control complex physical devices. Examples include digital computers, modern telephone switching systems, aircraft flight controls, and robotic manufacturing tools. The larger systems software applications, such as IBM's multiple virtual storage (MVS) operating system, are about 100,000 function points in size, which is equivalent to roughly 10,000,000 source code statements in common procedural languages such as Fortran, PL/I, or Ada.

Systems software is of similar size and complexity levels to many large-scale military applications. However the civilian systems software domain manages to build large applications with smaller specifications, shorter schedules, lower costs, and equal or higher quality than normally found on defense projects. The companies that build systems software tend to utilize sophisticated internal standards augmented by major international standards, such as those published by the IEEE and ISO.

Both the civilian systems software domain and the defense software domain excel in software quality control. Since both domains are concerned with complex hardware platforms that are controlled by software, it is imperative to have state-of-the-art quality control or the hardware devices may fail or perform in hazardous ways.

In contrast, the companies that produce information systems, commercial software packages, and software not controlling physical devices often lag in software quality control. There are both social and technical reasons for this. For example the systems and defense software domains almost always have formal quality assurance departments, while the information systems and commercial vendors are not as likely to have quality assurance groups.

Historically, the systems software and defense domains evolved from older engineering groups that had quality assurance support even before computers were utilized. The information systems domains evolved from accounting, finance, and business operations that seldom utilized quality assurance before computers were common.

Another aspect of the DoD attempt to move in a civilian direction is increased usage of commercial off-the-shelf software (COTS). Obviously the use of COTS packages refers to ordinary business and personal software packages such as databases, payroll programs, spreadsheets, and the like. The COTS concept is clearly not aimed at sophisticated weapons systems where no civilian packages exist.

Unfortunately, the military and defense domain have no strong incentive for adopting civilian best practices other than innate professionalism. The DoD itself and the military services are not profit-making organizations. If they tend to overspend or develop software in a way that is more costly than the civilian sector, so long as the fundamental mission requirements are not compromised, there is no overwhelming reason to improve.

For contractors, there are actually business reasons for staying somewhat inefficient compared to civilian norms. For time and materials contracts, there would be a negative incentive for adopting civilian best practices since increased productivity and shorter schedules would reduce the revenues and the profits from major contracts.

For fixed-price contracts, a case might be made that adopting civilian best practices would lower costs and raise the probability of gaining the contract. However, artificially low bids are common enough that this strategy might not be effective. The whole process of military procurement and contracting is in need of very careful analysis and possible re-work.

Defense Software Technologies

The military services and the DoD have been quite active in software technology research. Many initiatives have been funded and several prominent organizations, such as the Software Engineering Institute (SEI) and the Software Productivity Consortium (SPC), have focused much of their research on defense and military software.

Historically, the major programming languages used for military software included assembly language, Fortran, and some specialized languages that were seldom used outside of the military domain: Jovial and CMS2. The Ada83 programming language and the newer Ada95 programming language continue the tradition of developing specialized languages for defense software. However, the Ada languages have also attracted some civilian users, especially so in Europe.

Because of the diversity of software applications under the overall military umbrella, almost all programming languages are used. For example COBOL is used for more business-oriented military software such as payroll applications. The C and C++ programming languages are also used. Of the total of about 600 programming languages in current use, we have noted at least 75 languages on various military applications including JAVA, which is expanding in use among all software classes.

It is interesting that when productivity comparisons are restricted to coding, and exclude production of paper documents, the defense community and the civilian systems software community are roughly equal in terms of productivity. In other words, the defense programming community is as good as other software domains in coding.

Military projects often share common features with civilian systems software projects. One of these features is a need for high quality and reliability coupled with rather sophisticated software quality assurance groups. The military software domain has the second highest levels of defect removal of any type of software that Software Productivity Research has studied. Many military software projects top 95 percent in defect removal efficiency, and some have approached 99 percent. Since the U.S. national average is only about 85 percent in terms of pre-deployment defects removed, the defense community has had better than average quality control.

This is true for weapons systems and communications systems, but not necessarily true for ordinary defense applications such as payrolls and accounting that do not follow military standards. The domains that lag in software quality control include information systems, commercial software vendors, and some but not all outsource vendors. Of course in every domain there are broad ranges of performance, just as there are broad ranges in every human activity.

The military domain also ranks as number two in the use of software quality assurance departments. On many defense projects, more than 30 percent of the total work force is involved with testing and quality assurance tasks. Quality control in the military domain for weapons systems is quite sophisticated for obvious reasons. The main reason is because military software controls complicate physical devices such as radar sets and aircraft flight controls. If these do not work as intended, lives and battles could be lost.

The importance of quality control and formal processes within the military software domain explains why more defense software producers can be found at or higher than Level 3 on the Software Engineering Institute's (SEI) Capability Maturity Model® (CMM®) than other domains. Since many companies that are SEI CMM Level 3 produce both military and civilian software, there is some overlap between the systems and military software companies.

The defense software community deserves credit for pioneering software process assessments and process improvement technologies. The impact of the SEI's CMM has benefited many major defense applications and is spreading rapidly among civilian software producers as well.

In our studies since 1994, large applications of the same nominal size, such as 10,000 function points, appear to have better productivity and quality levels when produced by organizations at or above CMM Level 3. For smaller applications of around 1,000 function points in size, the data is less definitive but still favors the higher CMM levels.

A number of fairly sophisticated software quality approaches are quite common in both the military and systems software domains. The quality assurance and control methods used by both systems and military software include the following:

- Formal design and code inspections.
- Quality estimation tools.
- Quality and defect removal targets for key projects.
- Quality Function Deployment.
- *Six sigma* quality targets.
- Complexity analysis tools.
- Automated defect tracking systems.
- Test library automation support.
- Automated change control tools.
- Trained testing specialists.
- Formal regression test suites.
- Full life-cycle quality measurements.
- The SEI's CMM.

Both the systems and the defense software domains also strive for excellence in project management disciplines. Software cost estimation and software milestone tracking are very detailed activities in the defense domain. Software project management is an area where the defense community may be superior to most civilian sectors.

The military software domain utilizes the following two techniques that are seldom encountered on civilian software projects:

- Independent verification and validation (IV&V).
- Independent testing by a third party.

The phrase *IV&V* implies using a third party or an external company to investigate whether all requirements have been met and whether the design and other documents meet all relevant military standards. The phrase *independent testing* refers to hiring a company other than the prime contractor on a military software

project to conduct late stage testing after internal testing.

Both IV&V and independent testing add costs to military software projects that are not encountered on normal civilian projects. Whether or not these stages actually improved quality is ambiguous. It is true that military software defect removal is among the best of any kind of software project. However, it is no better than the defect removal found on civilian systems' software projects where IV&V and independent testing are not performed. Yet the military results are still better than those usually noted on information systems and commercial software applications, and on some outsource projects.

The overall results of the military software quality approaches have been generally successful. Indeed, only systems software and military software have approached or exceeded 99 percent in cumulative defect removal efficiency levels.

Large system development is inherently difficult and complicated. The defense software community often has a need for very large software systems that can approach or exceed 100,000 function points or 10,000,000 source code statements. At this large end of the spectrum, the defense community achieves better quality levels and more successful outcomes than any other domain except the best of the systems software producers. Productivity rates are fairly low, but failures and cancelled projects are low, too. Thus, the overall economic picture for building very large applications is not too bad in the defense sector. Indeed, for the largest applications beyond 100,000 function points, military software is an overall leader in terms of success and failure ratios.

It can be said that the strong emphasis in the military world on rigorous processes, complete specifications, and formal quality assurance controls produce projects that are fairly successful above 10,000 function points in size and even above 100,000 function points. These large software projects are expensive of course, but being able to complete such projects and have them work is a very difficult task. The success of the military software community on very large software applications is commendable.

Conclusions

Overall, the defense move toward civilian best practices is encouraging. However because this initiative only started in 1994 and required several years of research, the results may not be fully visible until sometime around 2005 or later. The reason for this is because applications in the 10,000 to 100,000 function point size ranges normally have development cycles approaching five calendar years. Thus, major defense applications using civilian best practices and standards are still under development and hence not yet studied and measured in terms of overall productivity and quality.

The following are some of the conclusions that we have reached from studying both civilian and defense software projects:

- Large applications above 10,000 function points or 1,000,000 source code statements require rigorous quality control and capable project management to be successful. Large applications that skimp on quality control and are careless with plans and estimates usually fail. If they do not fail, they will run late and exceed their budgets by notable amounts.
- The strong emphasis on quality control and project management disciplines associated with the SEI's CMM leads to a greater probability of successful completion than less formal processes for applications larger than 10,000 function points or 1,000,000 source code statements. Organizations at or above CMM Level 3 are more likely to be successful on large systems than those at Levels 1 or 2.
- For small applications below 1,000 function points or 100,000 source code statements, formal processes are not as significant as the experience of the development team. This is because teams are small so competence – or incompetence – of even one person tends to be visible and significant.
- For small applications below 1,000 function points or 100,000 source code statements, the level achieved on the CMM by the development team does not lead to major differences in successes or failure rates.

Overall, there are hundreds of ways to cause software projects to fail, and only a few ways to make them succeed. The highest odds of success will be found where capable teams use formal quality control and formal project management methods.

The military and defense community has been a pioneer in both quality control and project management methods. This appears to have paid off when building large software packages. Capable software personnel are in great demand everywhere so all domains are striving to select and keep good personnel.

The DoD's move to civilian best practices is encouraging and indicates a desire to improve software performance. Of course, quite a few civilian practices are of marginal value so one of the problems facing the defense community is to select practices that are truly best in terms of achieving high levels of quality, reliability, productivity, or other tangible factors.◆

References

1. Perry, William J. "DoD Policy on the Future of MILSPEC." CrossTalk Sept. 1994.
2. Sorensen, Reed. "Software Standards: Their Evolution and Current State." CrossTalk Dec. 1999.
3. Dempsey, Paul Stephen, et. al. Denver International Airport: Lessons Learned. McGraw-Hill. Mar. 1997.
4. Johnson, James, et. al. The Chaos Report. West Yarmouth, Mass.: The Standish Group, 2001.

About the Author

Capers Jones is chief scientist emeritus of Artemis Management Systems and Software Productivity Research Inc., Burlington, Mass. Jones is an international consultant on software management topics, a speaker, a seminar leader, and an author. He is also well known for his company's research programs into the following critical software issues: Software Quality: Survey of the State of the Art; Software Process Improvement: Survey of the State of the Art; Software Project Management: Survey of the State of the Art. Formerly, Jones was assistant director of programming technology at the ITT Programming Technology Center in Stratford, Conn. Before that he was at IBM for 12 years. He received the IBM General Product Division's outstanding contribution award for his work in software quality and productivity improvement methods.

Software Productivity Research Inc.
6 Lincoln Knoll Drive
Burlington, MA 01803
Phone: (781) 273-0140
Fax: (781) 273-5176
E-mail: cjones@spr.com

Securing Information Assets: Security Knowledge in Practice

Lawrence Rogers and Julia Allen
Software Engineering Institute

System and network administrators are an organization's first lines of defense in protecting critical information assets. They need a framework for organizing and selecting security practices that are easy to understand, describe, and implement. The authors propose the Security Knowledge in Practice (SKiP) method as a solution.

Critical information assets (systems, networks, and sensitive data) can be compromised by malicious or inadvertent actions despite an organization's best efforts. System and network administrators are on the firing line and even when they know what to do, they often do not have the time to take action; operational day-to-day concerns and the need to keep systems functioning take priority over securing those systems.

Administrators must choose how to protect assets. But when managers cannot prioritize critical assets and threats (as part of a business strategy for managing information security risk), then the protections an administrator offers will be arbitrary at best. Unfortunately, managers often fail to understand that securing assets is an ongoing process. They do not consider this factor when allocating administrator time and resources.

Most system and network administrators learned from their peers how to protect and secure systems, not by consulting published procedures that serve as de facto standards accepted by the administrator community – no such standards currently exist. Administrators are sorely in need of a structure for organizing, prioritizing, and selecting security practices that is easy to understand, describe, justify, and implement.

Tackling the Problem

The Security Knowledge in Practice (SKiP℠)[1] method was developed to organize security practices published on the Computer Emergency Response Team (CERT®)/CERT Coordination Center® Web site into a more process-based approach, departing from the more common problem-based approach. SKiP defines a cyclical process for establishing and sustaining the security of critical information assets such as the following:

- Systems running mission critical applications.
- Network infrastructure, including routers, hubs, and switches.

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

EVM and Software Project Management: Our Story

Walter H. Lipke
Tinker Air Force Base

The Software Division at Tinker Air Force Base in Oklahoma has used earned value management (EVM) methods for more than 15 years. These management methods have had significant influence in the improvement of software development and maintenance practices of the organization. This article, in a story-telling manner, describes the use of EVM for managing software, and how its system of management facilitated a natural evolution that led to recognition, awards, and more importantly, on-time, at-cost, quality software.

This story spans more than 20 years. There has been a considerable amount of excellent work performed by several dedicated, persevering people throughout this entire time span. Although our story covers two decades, it is not meant to imply that employing earned value management (EVM) for managing software requires an exorbitant time to implement. Our story is an evolution of practice caused by failure, a desire to do better, and outside influences upon our business.

In describing our use of EVM, I will cover the subject chronologically. Our beginnings cover a period of time from 1979 to 1985. The efforts to understand our process occurred from 1987 to 1989. Our period of significant, measured, process improvement was from 1989 to 1996. Then a period of evolving and refining our process is described, beginning in 1997 and continuing today.

Before we discuss EVM and its influence upon our software practices, I will introduce you to the Software Division's mission and its products. Tinker Air Force Base (AFB) is an Air Force Depot, which performs maintenance and modification to several weapon systems (including B-1, B-2, B-52, and E-3) and jet engines, including their avionics.

The Software Division supports the automated processes associated with the depot repair processes. Our products are Test Program Sets (TPS) and industrial automation software. For clarification, a TPS is used along with automatic test equipment to automate the testing process for an item requiring maintenance. A TPS consists of software, an electrical-mechanical interface, and instructions for its use. The portion of the division supporting depot maintenance has an annual revenue of approximately $40 million.

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

Trials and Tribulations of a Non-Geek Engineer

When I was in college, a little music group I played with recorded a humorous parody of a song in response to a fight that broke out at the end of a basketball game. As the evil team we played prepared to venture to our arena, the song got a fair amount of airplay on a local radio station. Knowing there were far more talented musicians who would never be on the radio, I was content to take my 15 minutes of fame and go home. Now, here I am, an environmental engineer writing a column for a software journal. The reason I am writing this is in response to the May 2002 BackTalk column "Week of the Geek." I found that I did not fit the mold of the geek very well. I am an engineer, but just where do I fit in?

While my dad and sister are steeped in high-tech engineering pursuits, I seemed to be destined for water and wastewater projects for the Air Force. Growing up an Air Force brat, my earliest introduction to these projects was venturing into wading pools at Wright-Patterson Air Force Base in my clothes to the point where my dad would have to retrieve me, often in his Sunday best or Air Force uniform. My earliest wastewater experience was losing a Snoopy in the sewer when my dad was stationed in Korea. Not to mention the countless number of times I set up Army men in the yard only to turn on the hose and flood them all out in a watery mess.

These used to be chances for me to get into trouble, but now I make a living at it. For example, on a recent trip to an overseas installation, as part of our study, we opened a fire hydrant, which ripped half the bark off of a tree by the wing commander's house. As a kid, I would have been sent to my room, been introduced to a belt, and told, "don't do that." As an engineer, I now go to a conference room and get introduced to the base civil engineer where I am told "good job."

This disconnect I have with software engineers was never more evident than the May BackTalk. The indicators of geekiness were well laid out, but I failed the test miserably. After some reflection, I determined that it is possible to be an engineer and a non-geek. After some painstaking research (conducted on several drives home), I came up with the following indicators of a non-geek engineer:

- You read Dilbert and wonder why everyone picks on the pointy-haired guy.

- You are shunned by the "Beta Geek" cliques.
- You read the assembly instructions for your kids' toys before you attempt to put them together.
- You identify the picture in the May 2002 BackTalk column as "T.J. Hooker looking at wigs."
- You tell the Wal-Mart associate in the computer department that you want a computer with a "Baud modem."
- You shop for computers at Wal-Mart.
- In reading the May 2002 BackTalk, you wondered if "X" was a variable for all "Dummies" books or an actual programming language.
- You think "Fry's" is a place to order unhealthy food.
- You did not buy Star Wars tickets until the day you actually saw (or will see) the movie.
- You think "TNG" is a copycat of the rap group "Run DMC" and has nothing to do with Star Track.
- You say Star Track.
- You have no computer disks of any kind on your person at this moment.
- When you see the sign that says, "Positive I.D. check in progress," as you approach the guard shack, you think the guard is going to say something nice about your driver's license photo.
- When you shop for a car, you are more concerned about gas mileage than gadgets.
- The cross talk you read is a journal for backflow prevention devices.
- You only subscribe to CrossTalk (the software journal) because your sister is the associate publisher.

Kevin Leachman, P.E.
Technical Project Manager
Trajen Systems
kevinleachman@excite.com

Can You BackTalk?

Here is your chance to make your point, even if it is a bit tongue-in-cheek, without your boss censoring your writing. In addition to accepting articles that relate to software engineering for publication in CrossTalk, we also accept articles for the BackTalk column. BackTalk articles should provide a concise, clever, humorous, and insightful article on the software engineering profession or industry or a portion of it. Your BackTalk article should be entertaining and clever or original in concept, design, or delivery. The length should not exceed 750 words.

For a complete author's packet detailing how to submit your BackTalk article, visit our Web site at <www.stsc.hill.af.mil>.

Get Back to the A-B-Cs of Software Management with the 2003 STSC Seminar Series

For the third year, the Air Force's Software Technology Support Center (STSC) is offering a series of informative software-related seminars in a workshop environment. This year's series will focus on some of the fundamentals of software management in acquisition and development programs, a Back to Basics.

The 2003 STSC seminar series will include these topics:

January 14-16	Lifecycle Software Project Management	Hill AFB Vicinity
February 18-20	Lifecycle Software Project Management	Hanscom AFB Vicinity
March 11-13	The Requirement for Good Requirements	Hill AFB Vicinity
April 22-24	The Requirement for Good Requirements	Hanscom AFB Vicinity
May 13-15	Software Cost Estimation	Hill AFB Vicinity
June 17-19	Introduction to CMMI	Hanscom AFB Vicinity
July 15-17	Introduction to CMMI	Hill AFB Vicinity
August 19-21	Software Risk Management	Hill AFB Vicinity
September 16-18	Software Quality Assurance	Hill AFB Vicinity
October 14-16	Software Acquisition	Hill AFB Vicinity
November 18-19	Bringing it All Together for the Software Manager (Software Best Practices: An Executive's Perspective)	Hill AFB Vicinity

These seminars are **FREE** to all U.S. government employees, however, seating is limited. So act quickly.

For additional information, visit our Web site at www.stsc.hill.af.mil

SPACE IS LIMITED. To reserve your place at any of these workshops, contact Debra Ascuena at 801-775-5778 (DSN 775-5778) or debra.ascuena@hill.af.mil.

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

Published by the Software Technology Support Center (STSC)

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

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