CrossTalk

November 2006 The Journal of Defense Software Engineering Vol. 19 No. 11

MANAGEMENT BASICS

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk: The Journal of Defense Software Engineering. Volume 19, Number 11, November 2006	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **OO-ALC/MASE,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

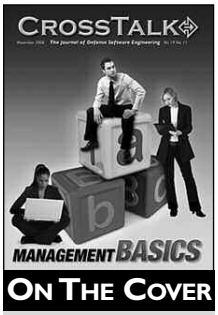
14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**32**	

Management Basics

Software Engineering Technology



ON THE COVER

Cover Design by Kent Bingham

Additional art services provided by Janna Jensen

Departments

CROSSTALK

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Diane Suchan*
DHS Co-Sponsor	*Joe Jarzombek*
NAVAIR Co-Sponsor	*Jeff Schwalb*
PUBLISHER	*Brent Baxter*
ASSOCIATE PUBLISHER	*Elizabeth Starrett*
MANAGING EDITOR	*Kase Johnstun*
ASSOCIATE EDITOR	*Chelene Fortier-Lozancich*
ARTICLE COORDINATOR	*Nicole Kentta*
PHONE	(801) 775-5555
E-MAIL	crosstalk.staff@hill.af.mil
CROSSTALK ONLINE	www.stsc.hill.af.mil/crosstalk

CROSSTALK, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF), the U.S. Department of Homeland Security (DHS), and the U.S. Navy (USN). USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection. USN co-sponsor: Naval Air Systems Command.

The USAF Software Technology Support Center (STSC) is the publisher of CROSSTALK, providing both editorial oversight and technical review of the journal. CROSSTALK's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 30.

517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

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: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CROSSTALK.

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

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

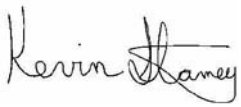
Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

A Willingness to Keep Learning

I believe the most valuable skill of a manager or leader at any level is their ability and willingness to learn. My wife calls this having a *teachable spirit*. Whether we are new to management or are a senior executive, we have to be willing to learn. I've taken lots of courses and seminars on how to manage and lead people but my most valuable teachings come from life experiences and studying other successful and unsuccessful leaders. Lessons that didn't come from textbooks or speakers have had the most influence on how I lead people. If there was one piece of advice that I could give to all new managers who want to be successful, it is to watch and learn. My first life lesson as a manager has stuck with me my whole career. I vividly remember when I first entered management. I had just completed a master's degree in management, and thought I knew what kind of leadership style best fit my personality and values. I quickly learned that one leadership style does not fit all people. Some people wanted to be empowered and left alone. Some people needed clear and daily direction. Some people needed more praise than others. In general, different people excelled or responded to different styles so I quickly learned the value of situational leadership. I think a similar thing can be said about how we manage projects. As large organizations, we tend to have rigid or one-size-fits-all processes. But we managers need to listen to our project leads and do what is going to make the project excel. It is easy to say that we have a process and we must follow it, but one-size-fits-all processes can be just as ineffective as one-size-fits-all leadership. Whether you are leading people or managing projects, the key to improving your contribution is your willingness to learn.

Let me leave you with a few applicable quotes:

- A leader who won't listen when his people tell him he is going the wrong way is destined for a head-on collision.
- A leader who doesn't learn to become a better leader might as well not be one.
- A leader who doesn't learn from his mistakes will certainly repeat them.



Kevin Stamey
Oklahoma City Air Logistics Center, Co-Sponsor

Basic Articles

This issue of CROSSTALK includes a broad range of topics in order to cover the needs of various software managers. We start this month with a comparison of management approaches for traditional and agile software development methods in *Are Management Basics Affected When Using Agile Methods?* by Paul McMahon. Next, Esther Derby shares her insights from working with numerous software managers in *Becoming a Great Manager: Five Pragmatic Practices*. In *Implementing Phase Containment Effectiveness Metrics at Motorola*, Ross Seider discusses a basic practice that I believe all software processes should include. Our final theme article by Michael F. Siok, Clinton J. Whittaker, and Dr. Jeff Tian discusses how to plan the number of defects being released to the customer in *Exposing Software Field Failures*.

For those wanting to delve deeper into management mysteries, we have David Lechner's article, *Software Recapitalization Economics* that provides useful formulas to help with decisions regarding sustainment of old software. If you are a fan of Walt Lipke's articles, you will find additional insights in *Earned Schedule: An Emerging Enhancement to Earned Value Management*, which he co-authored with Kym Henderson. While this issue targets software development and acquisition management, all software practitioners can gain insight from this month's CROSSTALK.

Elizabeth Starrett
Associate Publisher

Are Management Basics Affected When Using Agile Methods?

Paul E. McMahon
PEM Systems

Just how different is project management when using agile methods? The purpose of this article is to help readers understand the similarities and differences of traditional and agile project management approaches, as well as provide information that can help them decide if an agile – or a hybrid agile – approach might be beneficial. Related factors to consider when making decisions about using Agile, hybrid-agile, or a traditional approach, along with real project case studies, are provided.

Let us start with the basics. First, fundamental to project management is planning, monitoring, and controlling. Monitoring and controlling are achieved by executing a plan and taking appropriate action when a project deviates from that plan. This article focuses on the planning activity. I like to simplify planning for new managers by breaking it down into five easy steps: *What? Who? When? How?* and *How Much?*

Traditionally, a project management plan is developed at the start of a project to capture the answers to the first four questions. The plan is then used in the *How Much?* category to develop the cost, monitor and control the project, and communicate to the stakeholders what we are doing.

At the fundamental level, planning includes understanding *what* must be done (scope of effort), *who* needs to do it (staffing and skills), *when* it needs to be done (life cycle and schedule), *how* it is to be done (reviews, methodology, tools, meetings etc.), and *how much* it will cost (budget). These same five steps occur on both traditional and agile projects.

The *What*: Scope of Effort

Traditional Approach

Project planning includes scoping the work. Traditionally, this has been accomplished by first partitioning the effort through a work breakdown structure [1]. The intent is to break down the work into manageable chunks that can be monitored and controlled.

Figure 1: *Traditional Functional Approach*

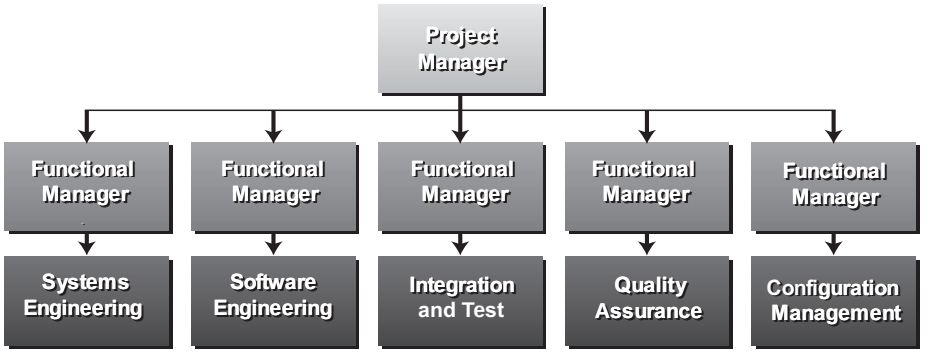
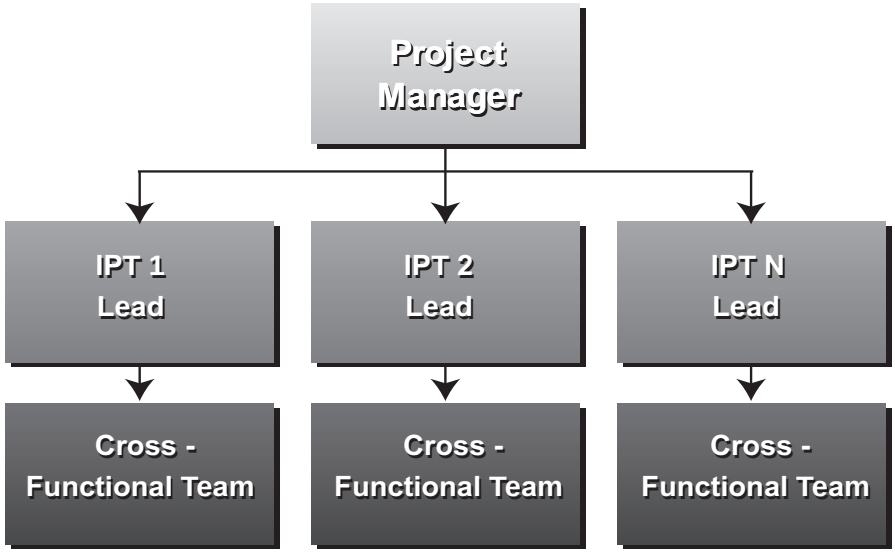


Figure 2: *Traditional Integrated Product Team Approach*



Agile Approach

The basic concept of breaking work down does not change with agile methods, but there is less detail provided early in the project and care must be taken in how work is structured so that it is fully scoped while potential solutions are not overly constrained. Some agile projects do not fully scope all the work up front. Scoping the work is discussed further in the section on *The When*.

The *Who*: Staffing, Skills, and Organization

Traditional Functional Approach

Traditional functional engineering organizations include systems engineering, software engineering, integration and test, configuration management, and quality assurance (see Figure 1). In the traditional, functional organization, tasking is through the functional manager, and the functional manager receives periodic status directly from assigned personnel.

Traditional Integrated Product Team Approach

Integrated product teams (IPTs) are cross-functional teams used in many organizations to achieve increased stakeholder collaboration and teamwork. Each IPT includes representation from all functional disciplines. Historically on large projects, IPTs have often been large teams (e.g. could have between 30 and 50 people on each IPT); therefore sub-IPTs may be formed for specific tasks. IPT tasking is usually through both the functional manager and the IPT, with the functional manager providing a higher level task definition, and the IPT providing project specific tasking (see Figure 2).

Agile Approach

The traditional functions are all still required when using agile methods, but the organization of the personnel, their interactions, and task reception may be different. The degree of difference depends partially on how your organization currently functions. Agile teams are small (usually no larger than 10 people each), and they are cross-functional like the traditional IPT. Two views of an agile organization are provided in Figures 3 and 4.

Figure 3 provides Agile Organization View 1, which has similarities to the traditional IPT structure. With this structure, agile teams operate as an extension of the traditional IPT similar to sub-IPTs in traditional organizations. Note the following caution associated with this view.

Not all organizations today implement true IPTs in the sense of *true* cross-functional and product focused teams. Many organizations have struggled with implementing IPTs because of the difficult culture-shift required from a functional perspective to an integrated product perspective. Similar issues are faced when implementing agile teams which are also cross-functional integrated product teams. Typical differences with agile teams from IPTs are their size (smaller), incremental approach to work, and visibility of tasks and task progress.

While Figure 4 appears significantly different from Figure 3, the real difference within a given organization may be more pictorial than real. Some have argued against representing agile teams as sub-teams of IPTs because it may give the impression they operate in a traditional, hierarchical, and functional fashion where tasking only flows down from management.

The *Hub* organizational structure [2, 3] depicted in Figure 4 is intended to represent the fact that the team is actively involved in its own task definition and estimation process, and proactively communicates with other teams directly when necessary. This is in contrast to the traditional, hierarchical organization where the focus of tasking and communication is through the chain-of-command. It is worth noting that this perception of how communication happens in traditional, hierarchical organizations is not always true.

This leads to the question: How different is the hub organization from the traditional organization? The answer to this question depends largely on the culture within your organization today. While the traditional organization structure represented in Figures 1 and 2 may seem familiar to many readers, implementations within specific organizations can be significantly different [4].

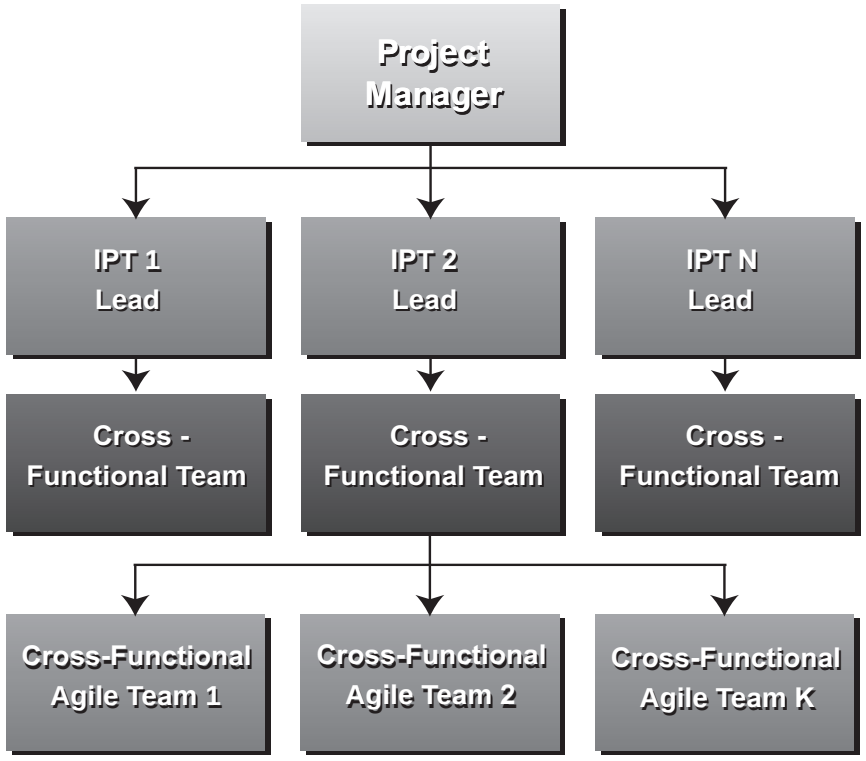


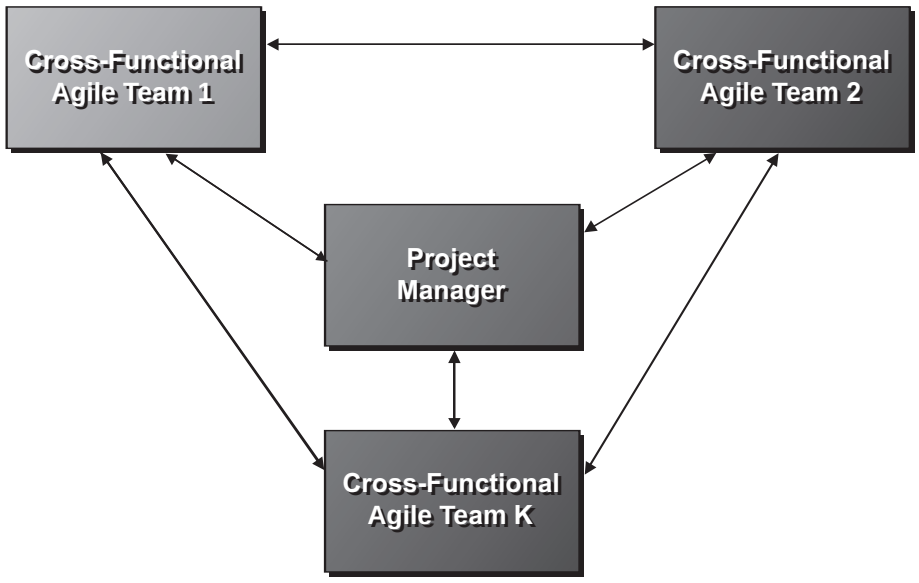
Figure 3: *Agile Organization View 1*

If an organization has effectively implemented *true* IPTs, and functional managers already have shifted their tasking to a higher level in support of project specific tasking by the IPT, then the shift to Agile may not be traumatic from a management perspective. This is because some of the hardest changes with Agile are cultural. If the culture is already *agile*, this makes the transition easier. In many organizations this operating model has existed for many years informally [4, 5].

Case Study 1

I have observed this condition in one of my client organizations that does not even refer to itself as being agile, yet they have exhibited agile-like behavior for many years. I have referred to this in previous publications as an unspoken adaptive (agile) subculture [5]. In this organization, small informal teams operate below-the-radar of the formal organization with a do whatever it takes team attitude to get the job done. Tasking from the functional managers in this organization is at a high enough level so that it is not an issue for a software engineer to help a systems engi-

Figure 4: *Agile Organization View 2*



neer and vice versa. It is expected as part of the job. The culture of what it means to do software engineering includes collaboration with systems engineering. I have referred to this in previous publications as the integrated tasking model [4].

Case Study 2

On the other hand, I have another client who employs what I have referred to as a *strict hierarchical tasking model* [4]. In this organization, systems engineers are tasked strictly and exclusively by the systems engineering functional manager (see Figures). In this organization it is not uncommon for a systems engineer to write a specification in her private office and then effectively *throw it over the wall* to the software engineering group with little personal interaction throughout the process. When a software engineer finds a problem in a specification from system engineering in this organization, they are not allowed to change the specification and the culture does not encourage software engineers to go talk directly to a systems engineer with respect to a potential change. They have to send it back through the formal chain, and it takes forever to get changes approved.

Different Reasons and Challenges When Moving to Agile

The organizations for both case studies use diagrams that look similar to Figures 1 and 2 to represent how they operate today. Both are interested in moving their organizations toward more agile practices. But the reasons they want to be agile and the challenges each face in making this transition are very different.

Why Would an Organization Want To Be Agile?

The reason Case Study 1 wants to move to Agile is for better visibility of work status and increased predictability. While this organization already exhibits a small team collaborative attitude, decisions sometimes get made without considering all the consequences. As a result, negative side-effects are not uncommon, such as slipped schedules and build instability. They are constantly at risk of falling into chaos and often rely on late-night heroics by individuals. The reason Case Study 2 wants to become more agile is because changes take far too long to cycle through all the bureaucracy in the organization, and they know they are not being responsive to their customer's needs.

Management Challenges Faced

In Case Study 1, a key management challenge is to get the small teams to self-manage their work more effectively. In Case Study 2, a key management challenge is to move decision making down in the organization and increase collaboration at the lower levels.

The *When*: Life Cycle and Schedule

Traditional Approach: Work Partitioning

Traditionally, work is partitioned into major functions, and those functions are further partitioned into tasks associated with requirements, design, code, test, and integration. Detailed schedules are developed up front in the project and task assignments are given to each developer.

Agile Approach: Work Partitioning

While the traditional functions are all still required, when they are done may be different. Up front, coarse grain planning is done [6]. This is high level planning for the long term of the project. This includes the high level requirements which are then allocated to increments. This is similar to traditional incremental planning, but with Agile, more details may be deferred which implies more planned collaboration with the customer later in the project.

As an example, one of my clients is modernizing a legacy system. We defined all the legacy system functional requirements up front, but we deferred details of the user interface. The most important thing to the customer was to move the functionality to the new system and shut down the legacy system as soon as possible. We did identify high level user interface requirements early as part of our coarse grain planning, but we deferred details because it was not high priority to the customer.

The customer collaboration on the user interface details needed to be planned. This collaboration was so important that we added it to the project schedule. This last step was also important to aid accurate progress reporting. Planned and scheduled collaboration throughout the project was a key difference with Agile methods.

Why Defer Details? Potential Advantage

The rationale for deferring details is based on the belief that by waiting to make decisions just in time, we have the best information possible and therefore reduce the chance of rework. Reducing rework means cost savings. Also, as we saw in the example of the legacy system, we may defer work based on value to the customer.

Why Defer Details? Potential Disadvantage

While on the surface this may seem logical, deferring details has also been known to lead to inaccurate progress reporting and scope creep late in projects. This is why it is important to place deferred work on the schedule or team task list where it is kept visible.

Recommendation to Help Minimize Scope Creep on Agile Projects

One recommendation that I have previously made [7] to help control requirements on an agile project is to fully scope the requirements up front at a high level (as we saw with the example of the legacy system) and then plan and schedule the time to work out the details at the start of each increment. If you have a good collaborative relationship with your customer this extra level of requirements control may not be necessary. But my experience on United States Department of Defense contracts has found more of an *adversarial customer-contractor relationship* than a collaborative one and therefore this extension to Agile for requirements control seems practical.

Agile Approach: Tasking and Scheduling Responsibility

While project scheduling and personnel tasking are still required with Agile, where responsibility lies and when a task is done may be different. Before making any scheduling and tasking responsibility changes, look closely at what your organization is doing today and analyze the effectiveness of the current process.

In some organizations, the traditional approach to scheduling is to build a large detailed schedule early in the project. The problem with this is that it can become difficult to maintain when changes on the project happen quickly. When this happens, senior management may no longer have an accurate picture of where the project truly is from a schedule perspective. If your schedules are accurately reflecting your project work, and you are able to keep them current, then it may not make sense to consider changing what you do. But if they tend to be difficult to maintain and often out of date then this may be a good area where some agility can help.

How Agile Can Help With Schedule and Task Status Accuracy

By keeping the project schedule at a higher level it becomes easier to maintain. This is not to say the details are not important. But by placing the details down inside each agile team and giving the agile teams the responsibility for keeping their status visible and up to date, one may be able to

increase their chances of accurate status reporting on projects.

A question often asked by managers considering moving toward Agile is *how difficult is this change going to be for my organization?*

The answer to this question usually depends on where your organization is today. In Case Study 1, the challenge faced was to get the team leaders trained in how to manage small teams more effectively and letting the team members know they are each responsible for maintaining and reporting their own task status back to the team. In this organization, there already existed some very successful small team leaders. The challenge was to mentor others in what the successful leaders in the organization were already doing so more of the organization could benefit.

Example of Management Change When Transitioning to Agile

In Case Study 1, the organization is gradually learning they need less detail in the high-level schedule as the small teams provide increased visibility into their task status. As more small teams in the organization meet their commitments, senior management's confidence grows, and they start to ask for less detail at the top. The change is not yet complete, and it is not happening over night.

Traditional Approach

Traditionally, when setting up work packages and planning the work to do, it makes good sense to use what has been referred to as a *rolling wave* approach. This means only plan work in detail for a short duration. When this short duration period gets close to the end, then plan the next wave in detail.

We used to do this 30 years ago when I was a programmer and a manager working for government programs building aircraft simulation systems. The *rolling wave* approach to planning is consistent with agile thinking today – plan the details incrementally and *just in time*. But over the years in some organizations, culture and control-oriented managers have pressured engineering organizations to plan excessive detail early.

Agile Approach

Agile approaches are driving us back to execute the rolling wave concept as it was always intended. But doing this the right way is not always easy, which is part of the reason why managers seeking to control their projects pushed for more detail early. When we do not plan the detail early, it becomes easy to abuse the process by pushing out real work that should be going on now. This potential downside of Agile needs to be taken into account when choosing whether to go with an Agile or a traditional approach.

The *How*: Tools and Motivating Teamwork

Traditional Approach

Traditionally, when establishing a plan up front, reviews to be conducted with the customer and internal reviews to the organization are identified. The methodology to be used is also established, along with planned tools.

Agile Approach

With Agile, reviews, methodology and tools must all still be planned, but the focus of the team shifts from the tools and methodology to personnel interactions concerning real project status. Tools can also affect the accuracy of status reporting as seen in the following case studies.

Case Study 3

At one of my client locations, Scrum [8] (a popular Agile method) was being used on a number of projects. The project leads and developers were reporting positive results with improved status reporting. The Sprint Backlog (Scrum term for team task list) was kept informally as *sticky* notes on the walls in conference rooms. Thinking it would improve the process, a commercial task management tool was introduced in the organization. Soon thereafter, team enthusiasm for the process and the disciplined and accurate status reporting fell off. After doing a little digging, I discovered that the commercial tool that had been introduced as a *process improvement* was not easy to use. The developers viewed the tool as a burden, which led them to stop updating their task lists and related progress in a disciplined way.

It is easy to look at this case study and say *tools can be fixed*, but too often they are not, which can result in inaccurate status reporting. But beware – there is another side to this story.

Case Study 4

A common practice on agile teams is to have team members sign up for work, rather than being directed to work on a task. The rationale for this practice is the belief that it promotes personal commitment to completing the task more effectively and on schedule.

My client, who does not refer to itself as agile (Case Study 1), uses a tool for task management that many in the organization do not like. All the developers in the organization are required to log into the tool every day to get their current assignments and report their status. People continually complain about the tool because of the time it takes to use it.

I would not say this organization applies self-directed team practices as described in many agile books, but they may still achieve a good part of the intent.

As an example, individuals are given tasks through the tasking tool by functional managers. They do not *sign up* for each task, but individual task performers do have an opportunity and are encouraged to communicate back with their manager after they have analyzed their task. If they do not feel they can meet the task due date, or they feel the allocated hours are insufficient, or if the task description is not clear, they can send the task back to the manager.

This communication back and forth usually creates healthy task discussions, driving an understanding of the real work being faced. As a result, increased visibility of where projects truly are is observed and senior management becomes aware earlier when projects are getting into trouble. There is also an improved customer confidence that has been observed as well by many throughout the organization.

It is worth noting that this organization-wide tool makes it easier for project team members, who are not collocated, to get their tasking and report progress, as well as participate on projects as team members from remote locations. It is also worth noting that sticky notes on conference room walls do not scale well especially on large distributed projects.

Motivating Teamwork

Because people sign up for tasks, and are asked to help others, an argument against self-directed teams has been, *why should I do my job and yours too?*

Jeff Sutherland, co-founder of Scrum, provided one idea how to do this through a performance review process he applies within his own company. It is a *weighted average individual performance review process* where the rating components include inputs from the customer, the team, and the company perspective. With this approach, employees can no longer get good reviews based only on the perceptions of their functional managers. Their review now depends on what their customer and teammates think, as well as their overall contribution to the organization. This technique can be used to help teams collaborate more effectively in both agile and traditional environments and it could help whether tasks are

assigned or *signed up for*.

The *How Much*: Cost and Metrics

Agile Cost

We do not yet fully know how cost is affected by employing agile methods. There is not yet enough real project data to draw conclusions. But with Agile it is a mistake to think it will cost less because we do less engineering. Rather, we partition the engineering work and do it at the best time, which should reduce rework cost. This can be done with hybrid agile methods and traditional methods as well.

Agile and Traditional Use of Metrics

Some organizations (both agile and traditional) use burn down (or burn up) charts [9], to indicate schedule progress. In organizations that follow Agile strictly, burn down charts are owned by the team, not functional managers. It is not a separate manager's view. But there is another side to burn down charts. Sometimes an objective perspective from outside the team can help, especially when team members lack progress estimation experience.

Ask yourself, who owns the burn down charts in your organization? Is it the team's perspective, or a separate manager's perspective? Is someone filtering the team's view? This is not necessarily bad. It might actually help convey the real progress more accurately, but it might not. It depends on your company's culture and project-specific conditions. But one indicator of whether it is working well for you is the accuracy of the reporting up the chain. Are you hitting your schedules? Do you have satisfied customers? Ultimately, these are the questions that you should ask to determine if your metrics and your status reporting system are working for you.

With Agile, the team members report their progress on the tasks they sign up to do. In *hybrid-agile* organizations such as Case Study 4, if they are given the task, they are also given an opportunity to discuss it with their manager and refine it, if necessary. When this process is working right, each day you can see the work to do – or team task lists – being updated, and work accomplished being checked off.

Agile Tailoring for U.S. Government Projects

Those who follow Agile strictly report progress on their task lists when code has been tested and works. In my recommendations to contractors on government projects, I tailor this strict software reporting focus to add all real tasks including documentation and preparation for customer reviews. My rationale is fueled by a desire for the burn down chart to represent *all* real work. When the burn down chart hits zero, I want to know that we are *really done*.

Agile and Traditional Use of Lines of Code Metric

Agile does not use lines of code as a measure of productivity, nor do many organizations using traditional methods. The problems with using lines of code as a progress (or productivity) indicator have been well documented, and with Agile this does not change.

First, this measure tells us nothing about value to meeting the customer's needs. It just tells us we have generated code. With Agile, the focus is on doing the simplest thing to achieve customer value, and it is viewed as more valuable to achieve it with less.

On the other hand, I have seen lines of code metrics used with traditional approaches as an effective trend indicator. For example, tracking the number of lines of code changed or added from one build to the next is a useful trend indicator, especially as a team nears a delivery milestone. It gives us a view of potential build stability and another perspective on how close we may really be to completion. The use of lines of code metric in this way is no different for an agile or traditional project.

Conclusion

The five basic steps of planning and controlling a project remain, but when employing Agile methods, these steps may be carried out with some key differences. Agile methods provide the opportunity for more accurate project status through self managed teams, and they also provide the opportunity for more rapid change processing. Case studies indicate that hybrid Agile-traditional approaches are often appropriate and can be particularly effective when based on the culture of a given organization.

Just how different project management is when using agile methods depends on the organization. It is not a matter of being agile or not being agile. There are many degrees of agility, and one can anticipate many decisions to be made along the way.◆

References

1. Project Management Institute (PMI). A Guide to the Project Management Body of Knowledge (PMBOK Guide). 3rd ed. Newtown Square, PA: PMI, 2000.
2. Highsmith, Jim. Agile Project Management. Addison-Wesley, 2004.
3. McMahon, Paul. "Lessons Learned Using Agile Methods On Large Defense Contracts." CROSSTALK May 2006 <www.stsc.hill.af.mil/crosstalk/2006/05/index.html>.
4. McMahon, Paul. Virtual Project Management: Software Solutions For Today and the Future. CRC Press, LLC, 2001.
5. McMahon, Paul. "Integrating Systems and Software Engineering: What Can Large Organizations Learn From Small Start-Ups?" CROSSTALK Oct. 2002 <www.stsc.hill.af.mil/crosstalk/2002/10/index.html>.
6. Cockburn, Alistair. Crystal Clear: A Human-Powered Methodology for Small Teams. Addison-Wesley, 2005.
7. McMahon, Paul. "Extending Agile Methods: A Distributed Project and Organizational Improvement Perspective." CROSSTALK May 2005 <www.stsc.hill.af.mil/crosstalk/2005/05/index.html>.
8. Schwaber, Ken. Agile Project Management With Scrum. Microsoft Press, 2004.
9. Cohn, Mike. User Stories Applied: For Agile Software Development. Addison-Wesley, 2004.

About the Author

Paul E. McMahon, principal of PEM Systems, helps large and small organizations as they move toward increased agility. He has taught software engineering, conducted workshops on engineering process and management, published articles on agile software development, and authored "Virtual Project Management: Software Solutions for Today and the Future." McMahon is a frequent speaker at industry conferences including the Systems and Software Technology Conference, and he is a certified ScrumMaster. He has more than 25 years of engineering and management experience working for companies, including Hughes and Lockheed Martin.

PEM Systems
118 Matthews ST
Binghamton, NY 13905
Phone: (607) 798-7740
E-mail: pemcmahon@acm.org

Becoming a Great Manager: Five Pragmatic Practices

Esther Derby
Esther Derby Associates, Inc.

Barry Boehm famously said, "Poor management can increase software costs more rapidly than any other factor." If that's true, then we should be attending as much to management practices as we do software development methods. In this article, the author describes five pragmatic practices that will help managers focus on the right work and create an environment for success.

I had lunch the other day with a fellow who had just pulled the kids out of school and left for a new job in a different city.

"What was it about the job that made it so attractive?" I asked.

"It was my boss," he replied.

What kind of manager generated that sort of loyalty? It was not a poor manager; poor managers create the illusion of progress with busyness. They depress productivity and morale. It was not an average manager – one who accomplishes work – but not always the right work. This fellow had a great manager, one he was willing to follow halfway across the country.

I have met and interacted with dozens of managers in software companies and information technology (IT) departments. Those who were successful accomplished goals that contributed to the bottom-line results of the company and developed people. They had sufficient domain knowledge to ask probing questions and understand risk. They understood the technology their teams worked with, even when they did not have the skills to implement the technology, and they all followed a similar set of management practices.

The following may not be practices taught in management schools or written down in software management and project management books. They are the common threads that emerge from my observations of successful managers.

Decide What To Do and What Not To Do

Great managers do not just accomplish work, they accomplish the right work. For companies that are in the software business, it is work that helps the company generate revenue, attract customers, and keep profitable customers. For IT shops, it is work that enables the business to operate effectively and efficiently.

Ask yourself how your company makes money and how your group contributes to the company's success. If your group develops the software that the company sells, the connection will be clear. Sometimes, it is not so clear. If you are in a testing group, you may contribute to business success by providing information that describes risks and enables good release decisions. If you are in a documentation group, excellent technical manuals will reduce calls to the support group and improve the customer's experience with the product [1].

Once you know how your group contributes to your company's success, you can articulate the mission for your group. For a development group, the mission could be stated: Deliver features that meet our customer's need and provide a return on investment for the company. When I managed a group that wrote software for investment portfolios, we determined our mission was: Provide accurate and up-to-date valuations for our funds.

We all have more than enough work to do; a clear mission helps us prioritize strategically important work and identify work that does not need to be done (at least not now, or not by our group).

I worked with a test manager to identify a mission for his group and prioritize all the work his group was doing. He defined his group's mission: Assess and communicate business and technical risks for the applications we test. As he listed the work his group was doing, one activity stood out: client site pre-sales technical support.

Two years earlier, he had helped the sales group by sending a tester on a sales call when a member of the sales team fell ill. The sales call was successful, and the sales manager continued to ask for testers to accompany the sales team on important accounts.

The test manager realized that while he was building goodwill, it was not in his mission to provide pre-sales support at the customer site. As he examined the work his group was doing (including the site visits), he saw that sending testers on sales calls was *preventing* the group from accomplishing work that would help assess and communicate technical risks on applications under test. He also knew he could not drop the responsibility immediately – without his testers, the sales team would flounder, and sales were the lifeblood of the company. He worked with the sales manager to provide training and transitioned to phone support only for calls.

Deciding what to do and what not to do helps focus efforts on the important work – work that will contribute to the bottom line of the company. Articulating a mission has another benefit: When everyone in your group knows the mission and how the work they do contributes to it, they will be able to make better decisions about their own work every day.

Limit Multitasking

It *seems* like assigning people to multiple projects will ensure all the projects are completed. But contrary to popular belief, multitasking does not speed work, it slows work and delays delivery. Multitasking creates the illusion of progress by creating busyness while robbing people of time and mental cycles. Humans are not particularly good at switching contexts [2]. Gerald M. Weinberg quantifies the amount of time lost in Table 1.

People lose time as they put away Task A and remember where they were on Task B. It takes time to retrieve and review documents or notes related to the task and to re-create a train of thought. Assigning multiple tasks buys time only when there are two tasks and one is clearly the top priority. The person works on the top priority task

Table 1: *Quantification of Time Lost to Multitasking* [3]

Number of Tasks	Percent of Time Spent on Each Task	Total Task Time
1	100	100
2	40	80
3	20	60
4	10	40
5	5	25

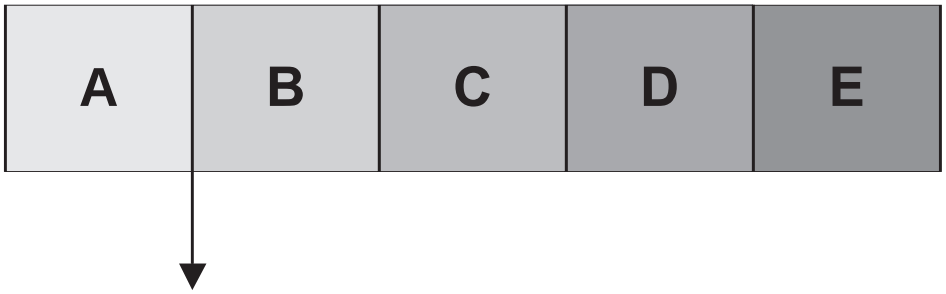


Figure 1: *One Person Assigned One Task at a Time*

until he is stuck or can go no further, then he picks up the lower priority task. This reduces time a person might spend spinning his wheels if he only had one task.

In product development, we can think of tasks as projects, or chunks of significantly different work. Two or more projects zap productivity. Suppose you have five tasks. Figure 1 shows how focusing on one task delivers value sooner. The arrow shows when the most important task, Task A is completed.

What if you have five tasks and only one person to do them? Rather than assign five tasks to each person, create a task queue. As each finishes a task she can sign up for another. This limits multi-tasking and will actually improve delivery time. Assume you have five tasks and they are all assigned to the same person, with instructions to make progress on all the tasks, because they are all important. The person doing the tasks splits her time between all five tasks. Figure 2 illustrates one simple effect of task switching. Each slice of pattern represents time spent on a different task. The down arrow shows the first time the project will earn value from a completed task. (In reality, the delay will be much longer because of time lost with each switch between tasks.)

Assume another project had identical tasks, but the worker focused on one task at a time until it was completed. The project would realize value much earlier, as illustrated in Figure 1. By the time the worker has completed one task, the focused worker has completed three tasks: A, B, and C. Plus, she has started on D [4].

In spite of the evidence, I often see people who are assigned to three or more projects. Managers expect people to spend a certain percent of their time on each project. In reality, the requirements of most projects do not fit neatly into 25 percent, 40 percent, or 75 percent of a week. The worst case is expecting people to switch between every assigned project every day.

In most workplaces, people are not actually working on project tasks a full eight hours a day. In a talk at the Software Development Best Practices Conference and Expo in 2003, Noopur Davis of the Software Engineering Institute reported that across 200 projects, the average amount of time spent on task (work that is directly referenced in a project plan) is 15 hours a week [5]. That is three hours a day! How much can a person really accomplish in 45 minutes – 25 percent of three hours – in each day? The largest slice becomes the de facto priority and everything else is squeezed and rushed into overtime hours when productivity is lowest.

Great managers realize this and assign work (or have people sign up for work) on one project or on two roughly similar tasks. Great managers realize that they will deliver value to the business faster if they finish one important project before starting on another.

Keep People Informed

People work best when they have the information they need. Great managers share what they know about the task and how the task fits into the goal of the project or organization.

People need information about their tasks, but they also need information about the big picture. Poor managers parcel out knowledge on a need-to-know basis. Good managers provide information about specific tasks and also explain how the work fits into the group's mission and the company's success. Being able to put a task into a bigger context provides motivation and helps people make better decisions.

I met one manager who kept an overall project plan to himself and only allowed people to see their assigned tasks. The people on the team had to go behind his back to see who was dependent on their work, who they needed input from, and where they needed to work together. This manager lost an opportunity for people to see their work in a larger context and find efficiencies and possibilities for collaboration. His actions communicated that he did not trust people and viewed them as *biological code producing units* rather than intelligent, creative people who were capable of planning and managing their own work.

People need information about their work. They also need information about events and priorities within their organization. When people lack information, they fill in the gaps with their worst fears and rumors. Rumor and conjecture consume an enormous amount of time in organizations. Communicate what you can, without revealing confidential personnel information or violating clearances or contracts. When you do not know something, admit it and communicate when you will have more information.

Managers need to be informed, too, about status, obstacles, and concerns. Serial status meetings (the kind where each staff member reports status one-by-one to their manager) save the manager time but wastes everyone else's. Typically, they only provide a fraction of the information a manager needs to know. Many important issues remain unspoken in this sort of status meeting. For example, people may be reluctant to admit they are struggling, reveal obstacles, or talk about their professional aspirations in a group meeting. That is information that a great manager needs to know. Hold regular one-on-one meetings to learn about the personal side of work tasks and stay in touch with professional goals. Save team meetings for interdependent work, group problem-solving, and team decision-making.

Provide Feedback

People need to know how they are doing at work. Do not assume that people know when they are missing the mark or what they are doing well. Great managers meet one-on-one with people regularly, every week or every other week to have the opportunity to give feedback when course corrections are small. One-on-one meetings also provide the opportunity to notice and appreciate something about each individual's contribution.

Make feedback specific, so people can act on it. I talked to one woman whose manager told her she was too nice. She was at a loss for what to do. Useful feedback describes behavior or results and states the impact – whether you want the person to change or continue a behavior.

I employ the following structure for giving effective feedback:

1. Create an opening.
2. Describe the behavior or result.
3. State the impact.
4. Make a request or engage in joint problem-solving.

The following is an example of what a great manager might have told the woman who was too nice:

1. I have some observations that I think

might be helpful to you. Is this a good time to talk?
2. I noticed in our meeting today that you agreed to all the requests from marketing for feature changes.
3. I know you are working extra hours and have a long backlog. What I see happening is that you are losing credibility with marketing because you agree to do work that will not be accomplished in the timeframe that marketing expects.
4. Let us talk about what we can do to manage expectations and do the most important work without burning you out.

Great managers do not wait until the yearly performance review to give feedback – that is a prescription for broken trust. If you want someone to be successful, tell him or her what needs to change as soon as feasible so that person can make adjustments. There is no sense in allowing sub-par performance or mistakes to continue; that just drives down productivity and morale for the individual and the team.

Develop People

Great managers know the career and professional aspirations of the people they work with, and they strive to help them meet those goals when they can and help people move on when they cannot.

A longtime employee wanted to move from testing to development. She took on as many technical testing tasks as she could and studied programming on her own time. Because she knew how to read code, she was an asset to the team in understanding how to design tests that did not depend on a graphical user interface. When she told her manager she wanted to apply for a job opening in the development team, he blocked her. "You are too valuable here," he said. The employee was gone within a month, taking a job as a developer at another company, and her team and her company lost a valuable employee.

Great managers talk to people about the skills they want to develop and the direction they want to go. They find opportunities to build skills and capabilities in the day-to-day work. The intention to develop a new skill almost always falls victim to a pressing deadline unless those intentions are planned into daily work. Finding development opportunities in daily work plans builds skill and loyalty. Even when people move on, they remember the managers who invested in them instead of just using their labor.

Another tester wanted to move into project management. The deadline for the project meant it was not feasible to send the tester to an off-site seminar. But together, he and his manager identified a small library of books and articles for self-study. His manager helped him apply his *book learning* as the tester organized and tracked his own work as a mini-project. He coached the tester to break tasks into one- or two-day chunks and define completion criteria. He showed the nascent project manager how to gauge his progress and think through his options when his rate of progress did not match his desired rate. By the end of the project, both the tester and the manager felt confident in moving the tester into a test lead role where he could organize and track his own work and that of two colleagues.

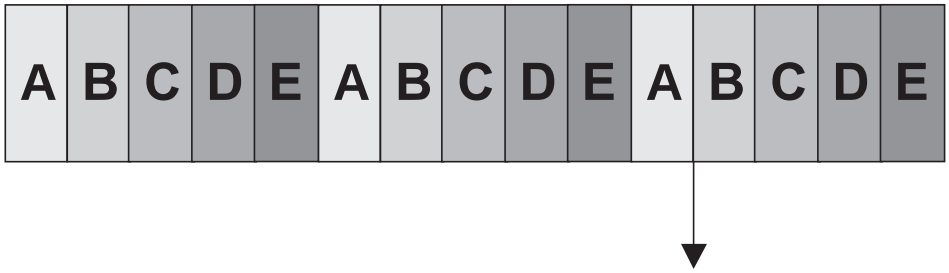


Figure 2: *One Person Assigned Five Tasks With Instructions to Make Progress on All of Them*

Conversely, when great managers do not have the work that will build skills, they tell people directly. Years ago, I worked with a young man who wanted to move from mainframe programming to PC programming. I knew that the work in our group would not help him reach his goals. Rather than try to hold on to him, I made a few introductions within the company. Later, after he had moved to a different department where he could do the work he wanted to do, he asked me out to lunch.

"You were honest with me," he said. "I felt bad about leaving the group, but grateful that you didn't try to talk me out of what I wanted to do." He became a good recruiter for me, too.

You do not need a big budget to develop people. Look for ways to develop technical or domain skills in daily work. Coach to increase organizational savvy and delegate to build management skills. It is one of the best ways to retain valuable employees – for you and for your organization.

Conclusion

Great management is not easy, but it is within reach. Apply these practices consistently, and you will be well on your way to being a great manager. You will leverage other's work by focusing on the highest priority work and enabling productivity. You will help people grow and develop by helping them see their work in context, by providing course corrections, and by giving them opportunities to develop new capabilities. And (maybe best of all) you will look good to your manager, too!◆

References

1. Derby, Esther, and Johanna Rothman. Behind Closed Doors: Secrets of Great Management. Raleigh, NC and Dallas, TX.: Pragmatic Bookshelf, 2005.
2. Shellenbarger, Sue. "Juggling Too Many Tasks Could Make You Stupid." CareerJournal <www.careerjournal.com/columnists/workfamily/20030228-workfamily.html>.
3. Weinberg, G.M. Quality Software Management: Vol. 1 System Thinking. New York. Dorset House, 1992.
4. Patrick, Francis S. "Program Management – Turning Many Projects Into Few Priorities With TOC." National Project Management Institute Symposium, Philadelphia, PA. Oct. 1999 <www.focusedperformance.com/articles/multipm.html>.
5. Davis, Noopur. "Self-Directed Teams: One Case Study." Pres. at Software Development Best Practices Conference and Expo, 17 Sept. 2003.

About the Author

Esther Derby is well known for her work in helping teams grow to new levels of productivity and coaching technical people who are making the transition to management. She is one of the founders of the Amplifying Your Effectiveness Conference and is co-author of *Behind Closed Doors: Secrets of Great Management*. Her latest book is *Agile Retrospectives: Making Good Teams Great.* Derby has a master's degree in organizational leadership and more than two decades experience in the wonderful world of software.

3620 11th AVE S
Minneapolis, MN 55407
Phone: (612) 724-8114
Fax: (612) 724-8115
E-mail: derby@estherderby.com

Implementing Phase Containment Effectiveness Metrics at Motorola

Ross Seider
On-Fire Associates

Phase Containment Effectiveness (PCE) is a project measurement technique that provides timely and accurate predictions of the project's current state and future risks. PCE methods deliver project insights beyond those provided by the Gantt and Program Evaluation Review Techniques (PERT). PCE provides data that increases the probability of a successful project outcome. This article describes the way in which PCE was implemented at one Motorola business unit.

Two of the most commonly used project management tools are the PERT and Gantt charts. When either is employed, a project's progress can be measured and extrapolations can be made about future project schedule accuracy. These tools focus on milestone achievement and task interdependencies.

The practical shortcomings of PERT and Gantt charts are well known to project managers. Among these are the following:

- **Milestone achievement can be ambiguous.** Often, milestones are qualitative events and opinions on attainment may vary. This ambiguity is not handled well by either tool. The project manager must rely on insight and experience to discern the true state of such milestones.
- **Neither tool inherently measures the quality of the milestone deliverable.** Nor do the tools anticipate downstream risks arising from poor quality of upstream deliverables.

Despite these and other limitations, PERT and Gantt techniques are popular because they do provide useful insight. Project managers can supplement these with other tools to gain further insight into a project's true state.

PCE is a quantitative, real-time measurement technique that addresses these limitations. PCE techniques provide an excellent means to unambiguously judge the completeness and quality levels of certain development milestones. PCE is an inherent indicator of downstream performance giving program managers time to adjust plans.

The PCE method described herein relies on the existence of foundation processes such as repeatable project planning, project tracking and oversight, and quality assurance. At the time of the events described in this article, the Motorola business unit was a Software Engineering Institute Capability Maturity Model® (CMM®)-Level 2 organization with many established Level 3 capabilities. For Motorola, PCE was a valuable next step in process maturation. Without the foundation processes, taking this step would not have been fruitful.

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

Theory of Phase Containment

It is a well-established project management axiom that the longer problems go undiscovered, the more costly they will be to correct.

For example, it is far less costly to correct an error in the architect's drawing than in the finished building. The architect's drawing error could have been discovered during a review of the building's design. It also could have been uncovered when the drawings were sent out to potential contractors. The building inspector might have detected the error before issuing the permit. Finally, it might have been spotted by the builders who reviewed the plans prior to the start of construction. A design error going unnoticed at all these review points may be very expensive to correct.

The thesis of phase containment is similar to the previously stated axiom: The most effective projects identify and fix mistakes at the earliest possible point in the project life cycle. This thesis is backed by substantial evidence across virtually all types of development activities. Stated another way: If mistakes are escaping early detection, then the project is at increased risk for missing its time and cost goals. Statistically, if the probability of escapes is increasing, the probability of costly and time-consuming downstream corrections is also increasing.

It was this thesis that convinced senior division management to sponsor a PCE improvement initiative at Motorola. PCE was an attractive initiative because it provided actionable management data that could further improve the effectiveness of the engineering operations.

PCE provides an early warning signal to diligent project managers. It allows them time to react; for instance, by adding resources to the system testing activities.

Applying PCE

As applied to software projects, PCE is a statistical measurement of the problems (bugs) uncovered at the earliest possible review (or containment) point compared to the total faults uncovered. The more effective the containment (the higher the PCE metric), the more likely the project will proceed smoothly and not experience downstream delays, quality problems, and/or cost overruns.

The PCE metric differentiates between problems discovered at the earliest possible review point to problems that are discovered at subsequent review points. The former problems are referred to as errors and the latter are referred to as *defects*. The sum of *errors* and *defects* represents the total project *faults* expressed mathematically:

Errors (problems discovered at the right time)
+
Defects (problems discovered at the wrong time)
=
Total Project Faults

Well-executed projects strive to discover a high proportion of the total faults as errors and not as defects. When properly conducted, project design reviews yield this type of data and the risk of downstream problems is reduced.

When faults are discovered during reviews, they must be classified as either errors or defects. This classification is made through a root cause analysis. Once faults are logged into the bug tracking database, the engineers and Quality Assurance (QA) team determine how and when each significant bug was injected into the design. If the root cause analysis determines that the fault should have been uncovered at an earlier review point, the fault is logged as a defect. If the root cause analysis determines that the fault

could not have been caught earlier, it is logged as an error. While there was incremental effort associated with this categorization, the Motorola development teams rapidly learned how to evaluate the sources of the faults, and the process took little additional time.

Implementing PCE served another Motorola objective: Management was committed to holding comprehensive design reviews. They were keenly interested in evaluating the quality and thoroughness of the design reviews. Phase-containment metrics provided data for this purpose, too. The fault information was combined with 1) the size of the work product being reviewed, 2) the number of people participating in the review, and 3) the time spent conducting the review. The combined data yielded benchmark ratios that allowed QA and project managers to evaluate the comprehensiveness of the review. Review milestones were challenged if the resulting data showed major variances from expected results.

Historical PCE Data

In principal, if a project is uncovering an *unusually large number* of defects, it is probable that additional latent problems exist and will be found downstream. But what constitutes an *unusually large number of defects*? Without a pre-existing comparison, this threshold is subjective. To solve this problem, Motorola reviewed historical fault discovery performance from earlier projects.

In many software organizations, new projects can bear strong resemblance to already completed projects. The resemblance can take many forms:

- The project is a variant of a past project.
- The project is similar in concept to past projects.
- The engineers executing a new project have worked together on previous programs.
- The software tools and processes are similar to the tools and processes used in the past.
- The new functionality is an add-on to the legacy functionality.

If two or more projects are similar, historical PCE data can add significant insight when interpreting new PCE ratios. Deviations from benchmark trajectories, either positive or negative, form the thresholds for comparison. When the current project's PCE ratios are compared to the norms measured on previous projects, negative deviations are a warning indicator. On the other hand, if the current project's PCE is better than historical benchmarks, then the project manager has evidence of reasonable progress.

PCE initiatives do not automatically require historical data reconstruction. Many companies choose to not implement PCE in this manner. Instead, they opt to accumulate historical data from new projects. While this approach is less burdensome, it takes longer to establish thresholds.

Case Study

Consider a hypothetical project named Reliant Release 4.0. Assume there have been previous releases of Reliant and the historic PCE records of past projects have been saved.

At an early milestone point in the project, the Reliant 4.0 development team created a Release 4.0 functional specification and a high-level design. Both of these have been design reviewed, and the results of the reviews have been captured in Figure 1.

The diagram's rows relate to the different Reliant 4.0 work products. The diagram's columns relate to the design review points. The spreadsheet's cells represent the data collected and analyzed at each review point. The four right-hand columns summarize the results of all the reviews by work product. The final column summarizes the total effectiveness of each review.

At the release of 4.0 functional specification review, six faults were uncovered. This number has been inserted into the chart in the upper left cell.

Sometime thereafter, at the high-level Reliant 4.0 design review, 139 total faults were uncovered. When evaluated for root cause, nine of these faults were attributable to Functional Specification problems (i.e. problems that should have been caught in the earlier functional specification review), and 130 were attributable to problems in the high-level design. Therefore, the numbers nine (the defects that escaped the earlier review) and 130 (the errors uncovered during the high level design review) are inserted into the second column. In a similar manner, as the Reliant 4.0 project continues, further data will be added to the chart.

Based on the data available at this early project stage, the PCE of the Functional Specification is equal to the ratio of the total errors/total faults. Mathematically:

PCE = # of errors / # of faults
PCE = 6/15 (40 percent)

This ratio is shown in the far right-hand side of the table's first row. Notice too that the PCE for the Functional Specification will degrade if further downstream reviews uncover new problems that are attributable to defects in the Functional Specification.

On one level, this PCE data suggests that a moderate amount of time may have been lost in high-level design implementing the wrong functionality. A more refined question to pose to project management is whether this situation is typical of the company's prior projects at a similar point in the development life cycle.

Given the availability of other project data, and the project manager's ability to assess the similarities and differences between Release 4.0 and earlier projects, Release 4.0's data can be put into a richer historical context. For the sake of illustra-

Figure 1: *Current Project*

Project Reliant Release 4.0

	Phase Review Points										
Work Products	Functional Spec Review	High-Level Design Review	Low-Level Design Review	Code Review	Module Test	Integration Test	Alpha Test	Total Errors	Total Defects	Total Faults	Phase Containment Effectiveness
Functional Specification	6	9						6	9	15	0.40
High-Level Design		130						130		130	1.00
Low-Level Design								0	0	0	N/A
Code								0	0	0	N/A
Test Plan								0	0	0	N/A
Faults by Phase	6	139									

Project Reliant Release 3.0

Phase Review Points

Work Products	Functional Spec Review	High-Level Design Review	Low-Level Design Review	Code Review	Module Test	Integration Test	Alpha Test	Total Errors	Total Defects	Total Faults	Phase Containment Effectiveness
Functional Specification	5	2	1	0	0	0	0	5	3	8	0.63
High-Level Design		78	34	5	15	11	12	78	77	155	0.50
Low-Level Design			112	32	22	13	11	112	78	190	0.59
Code				89	34	12	2	89	48	137	0.65
Test Plan					22	5	2	22	7	29	0.76
Faults by Phase	5	80	147	126	93	41	27				

Figure 2: *Benchmark Project*

tion, assume that Reliant Release 3.0 is a good comparative benchmark and that Figure 2 shows the phase containment data for the recently completed release 3.0 project.

One can observe in the right-most column that the PCE for Reliant Release 3.0's Functional Specification was 63 percent *for the entire duration of the project*. But at the time of Release 3.0's high level design review, (the point in time we seek to compare to Release 4.0's data), the Functional Specification's PCE was:

PCE = # of errors / # of faults
PCE = # of errors / (# errors + # defects)
= 5 / 7 (71 percent)

This data comes from the top row on the left-hand side of the Release 3.0 chart.

When comparing the new project to the benchmark project in their early stages, Release 4.0 is experiencing lower phase containment effectiveness than its benchmark project (40 percent compared to 71 percent). Even more glaring, the Release 4.0 Functional Specification PCE is already substantially lower than the *final* Release 3.0 Functional Specification PCE (40 percent compared to 63 percent). This should be a cause for concern on the new project.

Furthermore, the total number of faults discovered in the Release 4.0 high-level design review raises additional concern. This data is available on the bottom row of each chart.

High-Level Design Review Results:
- Release 4.0 139 faults uncovered
- Release 3.0 80 faults uncovered

This data raises further questions for project management, and possibly greater concerns. For example, is it reasonable that 59 additional faults should be discovered at this phase of Release 4.0 compared to Release 3.0? Were the two high-level design reviews equally thorough? Are there major functionality or complexity differences between the two releases? Is it possible that the Release 4.0 engineers have been exceptionally thorough and successfully uncovered a higher percentage of latent problems, thereby minimizing the escapes? Or is the data suggesting something more sinister?

Some of these questions cannot be conclusively answered at this point. However, the concern has been noted and the results of the next containment opportunity – the low-level design review – ought to be more closely monitored. If this upcoming review also reveals an unusually high number of escapes from the high-level design, one can reasonably conclude that Release 4.0 is getting into serious trouble.

If Release 4.0 milestones are tracking to expectations, PERT and Gantt tracking tools would not indicate any cause for concern. PCE metrics give the Reliant 4.0 project manager an early warning indicator.

By now, it should be obvious to the reader how phase containment metrics complement Gantt and PERT charts and how they better illuminate certain types of information. But one of the most important benefits is that this data is available early in the project's life cycle.

As organizations become better at collecting and archiving project metrics, their ability to accurately interpret phase containment data improves. Further root cause analysis and process re-engineering can address additional ways to improve operations, namely how to avoid the injection of faults. Ultimately, problem avoidance is the most cost effective way to speed a project.

Conclusions

Understanding the true state of an ongoing project affords organizations the opportunity to take timely corrective action. A project's successful outcome is best ensured by making adjustments at the earliest point possible. When Gantt and/or PERT metrics are combined with phase containment metrics, far richer project data can be discerned and much of the subjective problems with milestone-based tools can be avoided. If for no other reason than its early warning capability, PCE would have been a valuable addition to Motorola.

PCE's benefits extend beyond this. PCE metrics quantified the effectiveness of the business unit's design reviews. Improving reviews was a key element in Motorola's quality road map. Finally, over time and across multiple projects, PCE results provided quantitative evidence of continuous improvement. Project outcomes were more predictable and fewer faults escaped into customer environments.◆

About the Author

Ross Seider is president of On-Fire Associates, an executive management consultancy agency that focuses on engineering execution excellence. His career spans 35 years in Boston's high-technology engineering community. Previously Seider was vice president of product development and network operations for Cambridge-based Akamai Technology, and spent 12 years at Motorola in a variety of executive roles. In the 1980s, he co-founded and was vice president of engineering for two successful networking start-ups. Seider holds a bachelor of science in electrical engineering from Rensselaer Polytechnic Institute and a master of business administration from Northeastern University.

184 Brookline ST
Needham, MA 02492
Phone: (617) 680-3600
E-mail: ross@on-fireassociates.com

Exposing Software Field Failures

Michael F. Siok and Clinton J. Whittaker
Lockheed Martin Aeronautics Co.

Dr. Jeff (Jianhui) Tian
Southern Methodist University

Software Reliability Engineering (SRE) provides a way to quantify the likelihood of software failures in software-intensive systems during test and in-field operations. One simple-to-use, yet practical technique uses software reliability growth models, field and laboratory usage models, and acceleration factors to estimate software field failure rates. A case study illustrates the technique for fighter aircraft avionics software and compares predicted to observed software field failures. Establishing a target software reliability objective during software development and working toward it provides assurance that the software is achieving an acceptable operational performance mark – a mark that can be predicted, observed, and measured both in the laboratory and in the field.

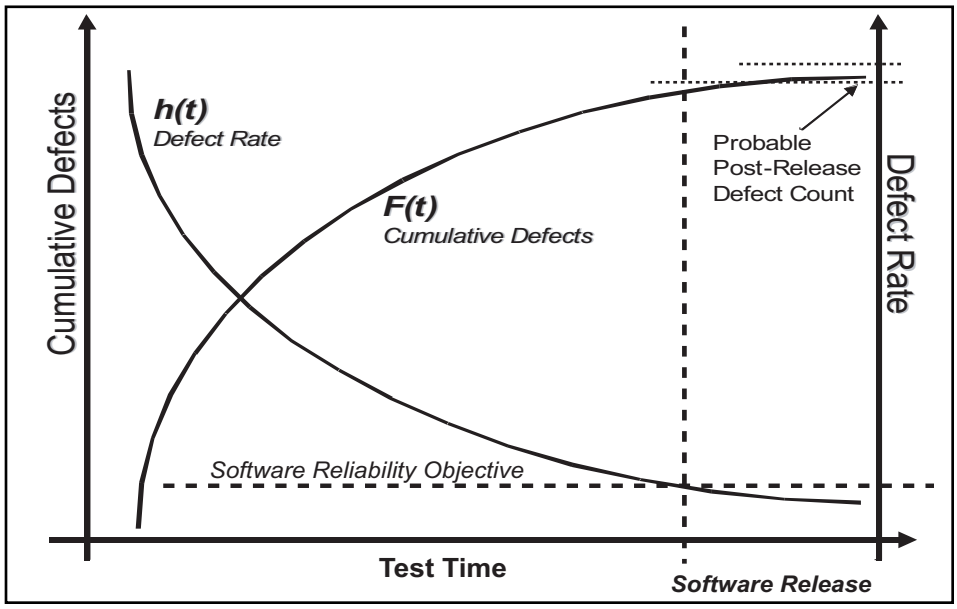
Growing the quality of software during a fighter aircraft avionics software development project is a natural outcome of conducting a disciplined software development effort. Through each step of the process, the software product evolves toward its end-product state, implementing more capabilities as it progresses toward its scheduled release date. Through a comprehensive and directed test regimen, the risk of releasing serious defects in the delivered software diminishes greatly as testing progresses, demonstrating software reliability *growth*. Once software development is complete and aircraft are being delivered, on-site operations teams manage the day-to-day operations of the deployed aircraft. Any malfunctioning avionic computers (e.g., computer will not boot, improper return to operation following a reboot, selected avionics operations in obvious violation of functional specification) may be managed through maintenance event actions by either replacing faulty equipment or reprogramming the computer and verifying a return to normal operation. These *software* maintenance events, however, have a cost – they take a fighter aircraft out of active service for the duration of the maintenance action. While software may not be the root cause of all such software maintenance events, if it were responsible for some, then perhaps models could be developed to predict their frequency. With these models, the software organization could then drive their software development efforts toward a specifiable and measurable operational quality goal and later observe the results of that effort in the field.

This article describes a practical and tested approach to using SRE time-based software reliability growth models to relate the release quality of fighter aircraft avionics software to the occurrence of software maintenance events in field operations. Using this approach, the frequency of expected software maintenance events can be estimated before the software is deployed to the field. This provides the software development organization with a key software quality metric, one with a customer-oriented view of the *operational performance* of the avionics software – a mean time to next *software* maintenance event.

Software Reliability Growth Models

Software is nothing more than a set of instructions and data derived from a software design used to control the operation of a computer system. Once software is written, it is tested in stages (e.g., unit, component, system) to identify and remove defects [1]. All contracted capabilities in software are tested and verified prior to delivery. However, it is nearly impossible with large embedded fighter aircraft real-time computer systems to completely and economically test the software in all possible operational conditions under all possible workloads. A comprehensive and successful test program will, however, sufficiently exercise the system, providing assurance that the risk of a serious computer system failure due to software will be acceptably low [2]. Assuming that the software organization manages their software processes in keeping with or bettering historical performance and assuming that these processes are demonstrably in control, then software reliability growth can be assessed using software reliability growth models (SRGMs). An SRGM is a mathematical expression of software reliability growth based on, typically, the detection rate of software failures during test. A software failure in this case is the manifestation of a software fault (i.e., a latent defect in the code) during execution of the software, creating an anomalous behavior that would be evident to the system user [1, 2, 3]. Numerous SRGMs exist in the literature; standards and

Figure 1: *Example SRGM*



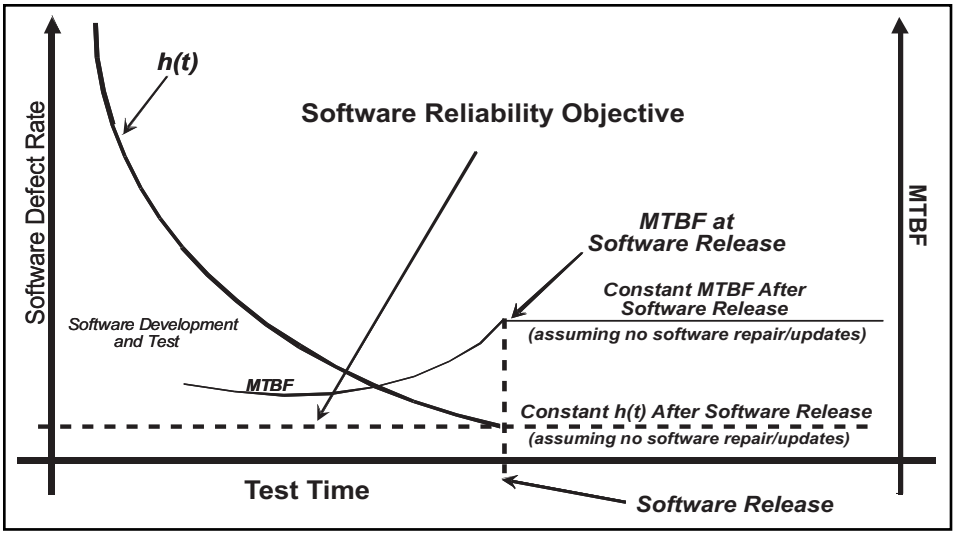


Figure 2: *Example Software Quality Goal Using SRE*

guidance on development and application of SRGMs, as well as other software reliability methods, are also widely available [2, 3, 4, 5, 6, 7]. While it is not necessarily a simple task, the software analyst chooses an SRGM that best models the defect discovery process on the software project while not creating an additional undue data collection and analysis burden on the software development staff [5, 7].

In its simplest form, an SRGM represents software reliability growth as a plot of defect arrivals per unit of measure (i.e., test time in this case) fitted to a mathematical model of the same. Defect counts may or may not be grouped. Software defects fall into the usual severity classes as their occurrences during test are visible by a system user and are recorded. Trivial, duplicate, and document-only defects are not counted.

Figure 1 (see page 15) provides an example SRGM; it illustrates the cumulative defects *F(t)* and Defect discovery rate *h(t)* curves. These curves show the expected arrival times and arrival rates of software faults discovered during test. Software reliability growth is also evident in this example – the rapid increase of defects discovered over time eventually starts to slow while testing continues at the same test intensity. By establishing a defect rate objective for software delivery and by planning the software development to achieve that objective, the software organization can estimate the total test time needed to reach that objective, estimate the probable release date of the software based on achieving that objective, and estimate the number of probable remaining defects in the software at release.

Software reliability objectives can be set early in the software development project so that software reliability growth can be modeled and tracked throughout the development effort. An example objective may be *no known severity 1, 2, or 3 defects at delivery*. This objective is quantified, coordinated, and is then, by declaration, an acceptable software defect rate at release. Figure 2 illustrates a software development effort progressing toward a software reliability/quality objective. A software mean time between failure (MTBF) metric may be computed and used as the measure of software release quality. *Failure*, in this case, means the occurrence of the next severity, 1, 2, or 3 defect[1]. This software MTBF measure is compatible with other system reliability measures used on the project outside the software organization.

Once released and deployed, software exhibits a constant failure rate (i.e., no more software reliability growth due to code corrections and re-release. For this discussion, assume no field updates for software). Since software is not updated in the field like it was during laboratory test, the software organization can easily render a prediction of the expected software field failure rate based on testing experience in the laboratory and expected usage in the field. However, field usage rates may vary significantly from laboratory usage rates causing observed field failure occurrences to differ noticeably from predictions. To render a useful prediction, software field usage needs to be quantified, compared to laboratory usage, and an acceleration factor must be derived.

Table 1: *Example Software Field Usage and Acceleration Factor*

Flight Month	(Flight Hours) FH	Number of Aircraft	FI	LUR	Acceleration Factor
1	227.6	13	17.50	90	5.14
2	314.3	13	24.17	90	3.72
3	521.4	16	32.59	90	2.76
4	550.3	16	34.39	90	2.62
5	591.4	16	36.96	90	2.43
6	652.0	16	40.75	90	2.21
7	590.7	19	31.09	90	2.89
.	.	.	.	.	.
.	.	.	.	.	.

Relating Laboratory to Field Experience

Software defect discovery rates at release in the laboratory are expected to be different from field-reported software defect discovery rates due primarily to the differing usage rates of the software in the various operational environments. Reliability engineers are aware of this phenomenon when working with electronic equipment, and they use an acceleration factor to adjust observed laboratory reliability performance with expected field reliability performance [8, 9]. The same technique can be used for software. The acceleration factor is simply a ratio of the laboratory usage rate (LUR) to the customer field usage rate [8]. LUR is the average usage rate of the software while it undergoes testing. LUR is determined by summing the software test time for each test system used during the test phases of the project and dividing by the average number of test systems used concurrently over the entire test period. The resulting LUR provides a usage rate that is similar to running the software in its various operational environments in x number of systems, where x is the

number of test systems instead of fielded aircraft.

To determine customer field usage rates, consider that deployed software may be installed on many fighter aircraft systems that may be operated within the same time periods but at different rates. Variable customer software usage must be counted for each aircraft. Flight intensity (FI) is used to establish an average customer usage rate for a group of aircraft. FI is determined by taking total flight hours (FH) and dividing by the number of in-service aircraft for each time period. The acceleration factor is then simply, LUR/FI. Table 1 illustrates how flight intensity and the acceleration factor are computed.

Estimating Software Maintenance Events

Estimating the expected number of software maintenance events is now a matter of completing a few simple computations. Starting with the software release quality expressed as MTBF, multiply by the acceleration factor to get a field-adjusted MTBF. Then, divide the expected total flight hours for that time period by the field-adjusted MTBF to get the expected number of software maintenance events for that time period. (Round the answer to the nearest whole number.) Table 2 illustrates how to estimate software maintenance events using software release quality, acceleration factor, and FH.

Process Summary

Figure 3 provides an overview of the process to compute the expected software maintenance events just discussed. To compute the software maintenance event rate, enact the following steps:

1. Determine the release quality of the software by reviewing software development historical records and modeling/analyzing software reliability growth. Determine also the LUR of the software; use models in absence of recorded data.
2. Quantify the expected customer field usage of the software. Determine FI using the number of in-service aircraft and FH. Use models in absence of recorded data.
3. From the customer and LUR, compute an acceleration factor to relate the laboratory software reliability experience with the expected field usage software reliability.
4. Compute the number of software maintenance events using software release quality, the acceleration factor, and FH. Identify and count maintenance events observed in the field that exhibit the characteristics of a software failure. Compare the observed software maintenance event data to predictions.

Flight Month	Software Release Quality ($MTBF_{SW}$)	Acceleration Factor	Field-Adjusted $MTBF_{SW}$	FH	Number of Expected ME (FH/Field Adj. $MTBF_{SW}$)
1	100	5.14	514.2	227.6	0
2	100	3.72	372.4	314.2	1
3	100	2.76	276.2	521.4	2
4	100	2.62	261.7	550.3	2
.	.	.	.	.	.
.	.	.	.	.	.

Table 2: *Computing Software Maintenance Event (ME) Estimates*

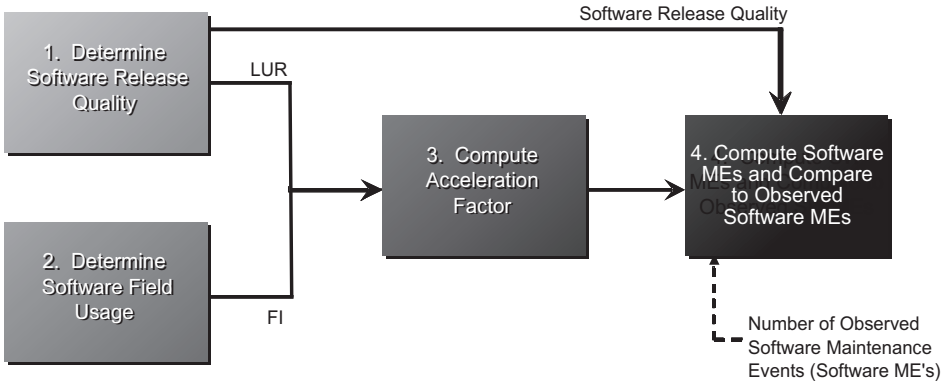
The following case study provides some practical experience using this technique.

Case Study

As part of an engineering study, the Lockheed Martin Aeronautics Company software engineering organization teamed with the system reliability engineering organization to determine if some aircraft field maintenance events reported with a particular aircraft avionics configuration could, in fact, be software or software-related maintenance events. Further, if they were, could their occurrence frequency be predicted? Recall that a maintenance event is a request for maintenance action on an aircraft due to an observed operational anomaly. To service the maintenance request, the aircraft is temporarily removed from active service. Once serviced, a description of the observed anomaly as well as the corrective action performed is noted on a maintenance event report and the aircraft is returned to active service. For this study, one embedded computer of a specific fighter aircraft avionics system configuration was identified for investigation.

Maintenance events are coded to identify an affected part needing maintenance action. *Software* maintenance events had no assigned maintenance codes. Suspected software maintenance events, however, were usually assigned to the *system* category against the embedded computer system serviced. Unknown maintenance event causes as well as a few other maintenance event *actions* were also assigned to this category. Software maintenance events and their subsequent actions had to be identified by reviewing every *system-coded* maintenance event record charged against the specific embedded computer. Selected other *non-system-coded* maintenance event reports were also reviewed looking for likely software maintenance events that might have been coded differently. Probable software maintenance requests and actions (e.g., software will not boot, improper avionics software operation after reboot, selected operations in violation of specification, computer reprogrammed and checked out OK) were identified and counted. From this work, the number of software maintenance actions per month was collected along with the number of aircraft and flight hours logged during the first 18 months of field operations of this specific

Figure 3: *Process to Compute Software MEs*



Flight Month	Software Release Quality (MTBFsw)	LUR	FI	Acceleration Factor (LUR/FI)	Field-Adjusted MTBFsw	FH	Number of Predicted Software ME	Number of Observed Software ME
1	100	90	17.50	5.14	514.2	227.6	0	0
2	100	90	24.17	3.72	372.4	314.2	1	0
3	100	90	32.59	2.76	276.2	521.4	2	1
4	100	90	34.39	2.62	261.7	550.3	2	1
5	100	90	36.96	2.43	243.5	591.4	2	2
6	100	90	40.75	2.21	220.9	652.0	3	2
7	100	90	31.09	2.89	289.5	590.7	2	2
8	100	90	33.24	2.71	270.7	638.2	2	2
9	100	90	33.73	2.67	266.8	683.7	3	3
10	100	90	26.75	3.36	336.5	699.0	2	2
11	100	90	14.87	6.05	605.0	491.9	1	1
12	100	90	15.17	5.93	593.3	703.9	1	2
13	100	90	14.68	6.13	612.9	689.2	1	2
14	100	90	18.47	4.87	487.2	867.1	2	3
15	100	90	17.49	5.15	514.7	830.0	2	2
16	100	90	12.27	7.33	733.2	582.6	1	1
17	100	90	12.85	7.00	700.4	616.8	1	1
18	100	90	15.12	5.95	595.3	725.6	1	2

Table 3: *Project Data*

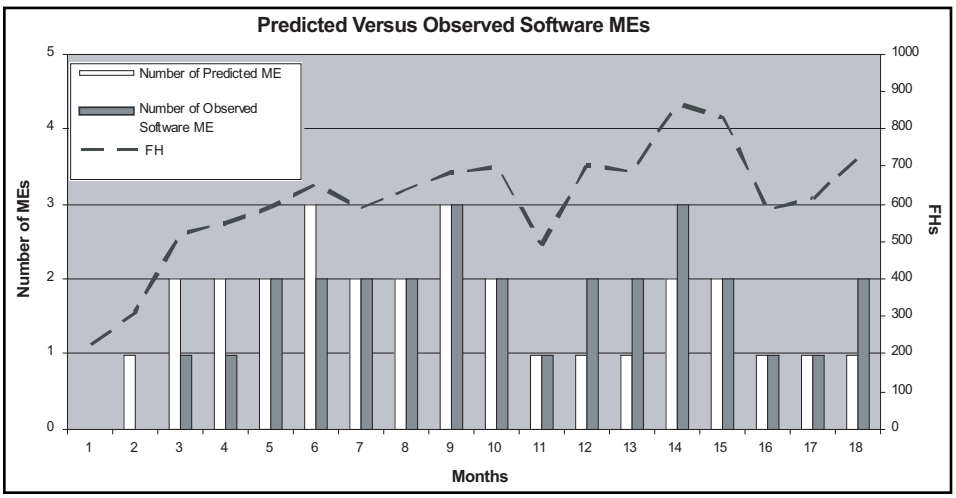
avionics configuration.

To establish the release quality of the avionics software, the management team on the project that initially developed and delivered the software was consulted on details of the software development activities. Software project metrics used during the development effort were reviewed to gain an understanding of the software process performance as well as the project dynamics at the time. From interviews, archived project data, and results of similar discussions on other similar in-house avionics software projects, an SRGM was selected, a LUR was estimated, and the software quality at release was computed and expressed as an MTBF.

The data for this engineering study from both the system reliability and software engineering groups was collected and organized into a spreadsheet. The software release quality, LUR, FI, acceleration factor, field-adjusted software MTBF, FH, and the number of predicted software maintenance events, rounded to the nearest whole number, were put into this spreadsheet. A last column identified the number of probable software maintenance actions observed in the data set. This data is presented in Table 3; the data is derived from the actual data taken from the study but has been altered to disguise proprietary information.

The number of predicted and observed software maintenance events for each month is plotted in Figure 4. FH is also plotted on the same graph. It was expected that the number of software maintenance events would follow the flight hours; that is, as flight hours increased or decreased for each month, so too did the number of software maintenance events. The observed software maintenance events exhibited this relationship with some small variation. In comparing the number of predicted to observed software maintenance events, the prediction tended to be a little pessimistic early in the field program, while later the prediction turned perhaps a little optimistic. This method models an effect generally observed in software reliability, where early in the field program when new software is released, field failure reports are initially high. As more systems are deployed with the same software, the number of failure reports per accumulated system time reduces to a near steady rate [3]. While not a perfect predictor of software maintenance events, this method does provide a useful approximation of the expected software field performance.

Figure 4: *Predicted Versus Observed Software ME Counts*



Some variations were expected in predicted and observed software maintenance event counts in this study. This variation could be attributable to the estimates of software MTBF, LUR, and FI used, possible errors in identifying appropriate maintenance event records for the study, errors in maintenance event record documentation, or record-keeping itself.

The study results indicated that, on average, the longer that the software was being run in this group of aircraft, the more likely it was that a software maintenance event would occur. It would seem to follow then that if one could predict the occurrence of these software maintenance events, one could also fix the software so that these events would not recur. Discovery of software maintenance event causes in the field is not usually easy, however. All the easy and nearly all of the *hard-to-find* software faults were found and fixed during the laboratory and flight-test program. Further, since maintenance event

reports are often short on problem descriptions and aircraft are not usually instrumented to collect data with the maintenance action request, the software maintenance events identified may not all *undeniably* be *software* maintenance events. Recovery and study of fault logs provide some help with fault cause identification, but require expert analyses. Without instrumentation data and/or complete descriptive information leading up to the point when the observed failure occurred, any likely future fix in software or the software development process is remote unless this same anomalous behavior can be repeated reliably in the laboratory or can be discovered in the laboratory on future versions of that same software.

To determine if these study results were coincidental, this same method was used on two other embedded avionic computer systems on two different fielded fighter aircraft programs. Not surprisingly, the results were similar. While results of three studies hardly prove method validity, results associated with these applications indicated that further use of the technique in-house was warranted and encouraged. Since this study was completed, SRE methods have been and continue to be applied on new start software development efforts and on additional completed and near-completed projects in a number of software application domains within the company. These SRE measures are being used in concert with other more traditional software metrics to estimate and predict the overall quality of the company-developed and fielded software.

A Note on Software Failure

When software fails to deliver its required service, the service is said to have failed. Software itself does not fail. Failures attributable to software generally occur because of requirements, design, or programming errors or they occur as a result of a computer equipment malfunctions. The software maintenance events recorded in this study could be attributed to any of these error types or computer equipment malfunctions. They likely resulted from any one or more of the following causes:

1. A specific rare event occurring in the operational environment uncovered a latent defect in the software.
2. A specific but rare use of the software not completely tested in the laboratory uncovered a latent defect in the software.
3. A specific unforeseen new use of the software not specifically provided for or tested caused an unexpected response from the software.
4. An unexpected rare event occurring in the operational environment caused an unexpected response from the software.
5. An equipment anomaly that the software was not designed to handle was encountered causing an unexpected response from the software.
6. Corrupted data caused anomalous operation.

Wrap-Up

With software size for fighter aircraft embedded real-time systems growing into the millions of source lines of code, implementing thousands of capabilities required to be operational under all workload scenarios, it is becoming a daunting challenge to economically assemble, debug, and verify this software within the complete range of system operations using current methods. Further, since software plays such a major role in providing the fighter aircraft's total system capabilities, the fighter's overall reliability is now being viewed as a function of the reliability of the hardware *and* the software [10]. Using the SRE methods described in this article to model the reliability growth of software provides a fairly simple-to-use and useful quality measure that can be used during software development as a predictor and estimator of software operational quality.

The SRE techniques described in this article provide a practical use of time-based SRE measures and metrics in identifying software product operational quality and in confirming that quality in field operations. It is a low-cost metric; it uses software fault and field failure data readily available to the engineering staff. In introducing this technique to our software engineering staff, no additional costs were incurred for data collection (i.e., required information was already being collected) and only a small cost was incurred for staff training and metrics deployment (i.e., about one person-month to develop training material and deliver instruction to a target audience of about 20 engineers). Our study results indicated that use of these few simple but tested SRE techniques within our current family of software measures had value as software operational performance predictors. Since this study, SRGMs have been studied and applied in-house on a number of fighter aircraft software programs. In the longer term, these measures are proving their worth and earning their way into our company's standard software practice.◆

References

1. Tian, Jeff. Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. Hoboken, NJ: John Wiley & Sons, Inc., 2005.
2. Musa, John. "Software Reliability Engineering: More Reliable Software, Faster Development and Testing." Software Reliability Engineering and Testing Courses. McGraw-Hill, 1998.
3. Lyu, Michael R. Handbook of Software Reliability Engineering New York, NY: McGraw-Hill Companies, 1996.
4. American National Standards Institute (ANSI). "Recommended Practice for Software Reliability." ANSI R-013-1992. Feb. 1993.
5. Carnes, Patrick. Software Reliability in Weapon Systems. Proc. of the Eighth International Symposium on Software Reliability Engineering – Case Studies, 2-5 Nov. 1997.
6. Schniedewind, Norman F. "A Recommended Practice for Software Reliability." CROSSTALK Aug. 2004 <www.stsc.hill.af.mil/crosstalk/2004/8>.
7. Wallace, Delores R. Practical Software Reliability Modeling. Proc. of the 26th Annual NASA Goddard Software Engineering Workshop, Nov. 2001.
8. O'Conner, Patrick D.T. Practical Reliability Engineering. 3rd ed. West Sussex, UK: John Wiley & Sons Ltd., 1991.
9. Ireson, Grant W., Clyde F. Coombs, and Richard Y. Moss. Handbook of Reliability Engineering and Management. 2nd ed. New York, NY: McGraw-Hill Companies, 1996.
10. Hamner, Robert S., "Software Reliability?" Transactions of the ASME Journal of Electronic Packaging 122. 4 (Dec. 2000): 357-60.

Note

1. The defect severity code identifies the consequence of the defect occurring in the released software. Severity 1 – Mission cannot be accomplished. Severity 2 – Task within the mission cannot be accomplished. Severity 3 – Task can be accomplished using a work-around.

About the Authors

Michael F. Siok is a software engineer for Lockheed Martin Aeronautics Company in Fort Worth, Texas. He has been with the company for 20 years, most recently performing software project planning and applying software reliability engineering techniques. Siok is a member of the Institute of Electrical and Electronics Engineers, the Association for Computing Machinery, and the International Council of Systems Engineering, and is a registered professional engineer in Texas. He has a Bachelor of Engineering Technology in electronics from Southwest State University in Marshall, MN; a master's degree in engineering management from Southern Methodist University (SMU) in Dallas, Texas; and is currently pursuing a doctorate in engineering management at SMU.

Lockheed Martin Aeronautics Co.
P.O. Box 748, MZ 8604
Fort Worth, TX 76101
Phone: (817) 935-4514
Fax: (817) 762-9428
E-mail: mike.f.siok@lmco.com

Clinton J. Whittaker has 18 years of professional experience as a reliability/avionics engineer. Currently, he is the reliability engineering lead at Lockheed Martin Aeronautics in Fort Worth, Texas. Since joining Lockheed Martin in 2002, Whittaker has provided reliability oversight of selected aircraft programs primarily focusing on avionics hardware and software development and test. His background includes reliability and avionics experience with Alcatel in telecommunication networks; Beal Aerospace, as the avionics lead over the hardware and software development for a commercial rocket venture; and Texas Instruments/Raytheon, as a production and field reliability engineer on two missile programs. Whittaker has a bachelor's degree in electrical engineering from Texas A&M University.

Lockheed Martin Aeronautics Co.
P.O. Box 748, MZ 2462
Fort Worth, TX 76101
Phone: (817) 762-3034
Fax: (817) 655-7181
E-mail: clinton.j.whittaker
@lmco.com

Jeff (Jianhui) Tian, Ph.D., is currently with the computer science and engineering department at Southern Methodist University in Dallas, Texas, and worked for IBM Software Solutions Toronto Laboratory from 1992-1995 as a software quality and process analyst. His current research interests include software testing, measurement, reliability, safety, and complexity and application to various systems. Tian is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery, and is a registered professional engineer. He has a bachelor's degree in electrical engineering from Xi'an Jiaotong University, a master's degree in engineering science from Harvard University, and a doctorate in computer science from the University of Maryland.

Southern Methodist University
Dept. of Computer Science
and Engineering
PO Box 750122
Dallas, TX 75275
Phone: (214) 768-2861
Fax: (214) 768-3085
E-mail: tian@engr.smu.edu

More Online from CrossTalk

CrossTalk is pleased to bring you these additional articles with full text at <www.stsc.hill.af.mil/crosstalk/2006/11/index.html>.

Integrated Quality Assurance for Evolutionary, Multi-Platform Software Development

Dr. Robert B.K. Dewar
AdaCore

A software product is rarely a static artifact resulting from a one-time effort; it needs to evolve. The development team might be distributed geographically, requiring careful coordination. A software producer must have well-defined processes for dealing with these issues, to ensure that its products successfully meet users' needs. This article is a case study of how one software producer, AdaCore, handles this challenge, but the processes are not company specific and can be adopted by any organization. By way of background, AdaCore is an international company that produces and supports a family of Ada language tools comprising compilers, debuggers, integrated development environments, supplemental tools and packages, and other components. With two releases of each product annually on more than a dozen platforms, the company requires clearly defined procedures for change tracking, configuration management and quality assurance. In many ways, AdaCore faces the same issues as other growing software organizations. This article shows how these are addressed through a combination of automated tools and human management.

Uncommon Techniques for Growing Effective Technical Managers

Paul E. McMahon
PEM Systems

This article utilizes real project scenarios to demonstrate a set of techniques that support common patterns employed by many effective technical managers across a range of organizations. Planning, status, metrics, and communication with task performers and senior management are addressed. If you are in a growing organization that is striving to institutionalize its processes, this article will provide you with a wealth of insights and practical techniques that could help you become more effective in your job.

Software Recapitalization Economics

David Lechner
WeC2 Technologies

This article analyzes the economics of cyclic replacement or recapitalization of software. It analyzes some of the historic issues and costs of software maintenance. Then, it provides analysis on how much software modernization occurs while still maintaining old code. I recommend a combination of minimal maintenance and a cyclic re-engineering to maximize the amount of code refresh and minimize the code's average age. In theory, regular refresh will allow rapid introduction of new or improved functions. It finishes by recommending strategies to target improved productivity as a component of recapitalization.

Large commercial and government systems are increasingly software intensive. The costs of supporting software over a 10 to 20-year lifetime are increasingly significant in today's software dependent world. Software support costs continue long after development ends, typically costing between 67 percent and 80 percent of the overall life-cycle cost. As shown in Figures 1A and B [1, 2], this is two-to-four times the development cost. This article will investigate options on how to spend that life-cycle maintenance budget to both maintain and enhance the code at the same time.

The U.S. Air Force (USAF) [2] studied 487 commercial software development organizations to see how software support costs are distributed among different tasks. According to the report, most software support dollars are spent on defining, designing, and testing changes. Support activities they identified include the following:

- Interacting with users to determine what changes or corrections are needed.
- Reading existing code to understand how it works.
- Changing existing code to make it perform differently.
- Testing the code to make sure it performs both old and new functions correctly.
- Delivering the new version with sufficient new documentation to support the user/product.

The USAF also shows that in post-deployment software support, 75 percent of the effort involves *enhancement* and refinement, and the remaining 25 percent is associated with *maintenance* of existing modules, as shown in Figure 2 (see page 22) [2]. Historically, the maintenance of software involves correcting bugs and supporting the required deployment changes. Research by Roger Pressman in *Software Engineering: A Practitioner's Approach* also confirmed this 21 percent factor and found that:

> … only about 20 percent of all support work is spent fixing mistakes, while the remaining 80 percent is spent adapting existing systems to changes in their external environment, making enhancements (possibly categorized as refinement) requested by users, and reengineering an application for future use. [4]

We will use the term *enhancement* for all non-maintenance program improvements, including refinement. We will use the term post-deployment support to include both maintenance and enhancement.

According to the USAF and other research, newly developed software has a high failure rate until the bugs are worked out, and after this point, failure rates usually drop to a low level [1, 2]. Theoretically, software should stay at that *reliable* level forever because it is not getting physical wear. Since software continues to get changed, however, it will continue to have bugs and reliability problems. *Thus, software wears out because it is maintained* [5].

There are more modern reasons for software maintenance, namely the following:

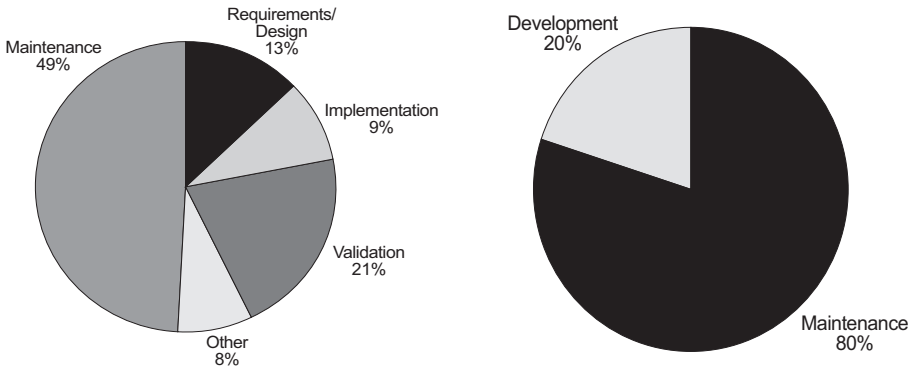
- Changes required for security considerations and protection from hackers.
- Changes in the operating system (versions), often driven by security or reliability.
- Changes due to other changing commercial off-the-shelf (COTS) software or hardware products.

Validating the Maintenance Expense

The 2:1 relationship of software support cost to development costs (i.e. 66 percent to 33 percent ratio) is critical to developing a strategy for life-cycle maintenance, but how real are these *typical* relationships? One significant study published data on coding errors and support [6]. The relevant data in Table 1 (see page 22) was taken from the Information Technology department of a large commercial bank. This data represented 35 projects and 20 million source lines of code (SLOC). We tested those rules of thumb by using them to calculate the theoretical support costs and compared them to the actual reported annual support cost.

The *age* of code was not available, so we assumed a mean 30 year lifetime and a *typical* productivity of one line of code per hour at a cost of $50 per man hour to obtain a derived development cost. That derived development cost was then multi-

Figures 1 A and B: *Software Life-Cycle Support Cost for Two Types of Large Programs*



Note: The USAF report used the term *maintenance* for generic post-deployment life-cycle support [2].

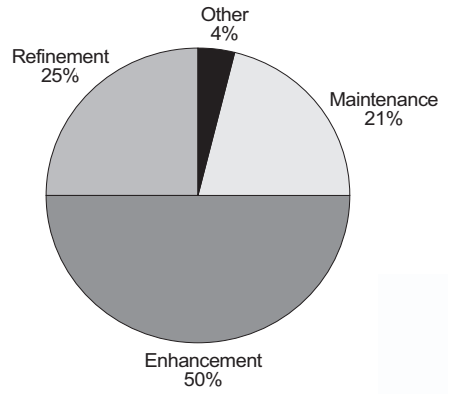


Figure 2: *Approximately 75 Percent of Software Post-Deployment Support Is Associated With Refinement and Enhancement to the Program* [3, 2]

plied by two and divided by 30 years to get a theoretical annual support cost, which we compared to the actual expenditures by the bank. This ratio was within 3 percent for the mean project value and 7 percent for the average project. This calculation does not correct for the present value versus annual value of money or the actual age of each program. The *rules of thumb* on software life-cycle maintenance however still seem reasonable and applicable today.

Given that the life-cycle software support costs is twice the development cost, we can divide that by a nominal 20 years lifetime to estimate annual support costs. We use 20 years here instead of 30 years since technology is moving quickly and we want a conservative estimate. This ratio implies that support costs 10 cents per year for each development dollar spent. This annual expense does not cover the capital recovery of the initial development expenses. We call this the *10 percent rule for software maintenance budgeting*. This rule checks the adequacy of a project's long-term software support budget, or perhaps determines the required sales/licensing revenue to justify a commercial project. To have a really successful long term project, with functionality growth and satisfied users (and perhaps a sustaining business operations point), the ratio of a long term support budget (or revenue) vs. a development budget should be at or above 10 percent. To recover development costs *and* do maintenance, the product income stream must be even higher (with a three year payback on development costs, the annual revenue must exceed 40 percent of the development budget).

This ratio illustrates the problem with starting up new software projects on a regular basis (such as Department of Defense Small Business and Innovative Research projects) unless they replace old code. The project is a distraction that siphons budgets from other maintenance needs unless the customer is prepared to spend 10 percent of the development cost on an annual basis to support the code. If the vendor does not commercialize to recoup the difference, the project is doomed.

This 10 percent rule can be dissected further. Data shows that 25 percent of costs are for maintenance and *other*, and 75 percent of the long term cost is for *refinement* and *enhancement* [3]. Thus out of our 10 percent of development expense, there is an annual 2.5 percent minimum budget required for reliability and bug fixing and 7.5 percent recommended for functionality change.

The High Costs of Software Support

Unfortunately, software support cost for old code, measured as a ratio per line of source code, can be up to 40 times more expensive than engineering new code [1]. This factor has a huge impact on long-term software support. It is commonly attributed to several reasons, namely the following:

1. The support engineers are often not the original developers and must relearn the program's design and requirements.
2. The code becomes more complex as it evolves and is patched, introducing more bugs and requiring more testing [7].
3. The programming technology is old and static, requiring greater labor.
4. Documentation problems require additional effort and repair.
5. There are definitive system limitations on how fixes can be made.
6. Extensive testing to ensure that the system performance is met [8, 9].
7. The use of older technology requiring older (more senior and more expensive) workers, often considered *gurus*, who are the only ones with the familiarity that can make a change [10].

Typical legacy style support involves analyzing user problems and requests, operating system updates, COTS product version changes, urgent reliability problems and security issues, and hardware obsolescence issues in order to define and determine what support is done. This effort can be challenging since some problems are incredibly difficult to find or recreate. Teams supporting legacy code typically work in a cyclic mode based on budget availability.

Software Re-Engineering

Software re-engineering is the complete overhaul of a software application, tearing it down to its component requirements and rebuilding it with modern methods, codes, and practices. The USAF policies advocate re-engineering of software *when the program manager concludes that it is better to pay now, rather than waiting to pay much more later* [11]. *Paying now* is what William E. Perry calls *avoiding the rathole syndrome*. Perry defines a rathole as *the dark place where software maintainers throw their money with no possibility of return on investment*, and he compares software with old cars. In the short term, it is cheaper to fix your old car than it is to buy a new one, but in the long term it costs more than buying a new car. Perry also explained that once one software rathole is plugged up (bug is fixed) another usually appears.

The 40:1 ratio of costs for maintaining old vs. new code should really be a factor that changes over time, starting out closer to a regular 1:1 factor if the maintenance team was chosen from the remnants of the original developers and used the same development environment. It would then worsen as the staff changes, tools age,

Table 1: *Recent Commercial Data Helps Validate That Post-Deployment Life -Cycle Costs Are Twice Development Costs*

Commercial Application Systems Profile (35 Systems, Total 20M SLOC, 1970s to 1990s) [6]				
Variable	Mean	Average	Minimum	Maximum
Errors per Month (After Deployment)	8.3	9.7	1	101
Support Costs/Year	$693,000	$661,000	$83,000	$3,532,000
Application Size – in # of Modules	217	266	18	1,500
Application Size – Lines of Code	215,000	185,000	54,000	702,000
Authors Analysis (Added):				
Rule of Thumb Development Cost	$10,750,000	$9,250,000	$2,700,000	$35,100,000
30 Years of Support Would Then Cost	$20,790,000	$19,830,000	$2,490,000	$105,960,000
Percentage Support Cost of Total Life Cycle Cost	66%	68%	48%	75%
Dev. Cost * (2/ Life)	$716,667	$616,667	$180,000	$2,340,000
Ratio of Theoretical to Actual Support Cost	103%	93%	217%	66%

complexity increases, and reliability degrades due to change [8, 9]. Some researchers identify code that is over 15 years old as *alien* code [8, 9], since by this age there are no longer any members of the original development staff left. Today, most applications that are old are unstructured, have architecture problems, poor documentation, and questionable change records. A good support strategy should avoid letting code get too old.

The *average equipment age* is a concept used in economics, replacing a capital item on a regular cycle is called *recapitalization*. At any given time the average age is the sum of the different ages multiplied by a weighting factor that is the inverse of the cyclic time period. In the example below, we shortened the math using the formulae for a series summation developed by Carl F. Gauss in the 1800s. The average age of an inventory just before one unit is replaced is simply half of the sum of one plus the cycle time period.

7 Year Cycle: (1/7)*1 + (1/7)*2 +(1/7)*3 + (1/7)*4 +(1/7)*5 + (1/7)*6 + (1/7)*7
= (1/7) * {1+2+3+4+5+6+7 }
= (1/N) * {Summation 1:N}

Generalized Case; Average Age of Items Replaced Every N Years:
= (1/N)*{N * (N+1)/2} by using Gauss' summation formula
= (N+1) / 2

Can we apply a cyclic recapitalization strategy to get code improvement via re-engineering and still continue acceptable support? To answer this question we analyzed a large hypothetical project of 20 million source lines of code. The key consideration is the amount of new software source code (new SLOC) created each year. One key input is the mix or allocation between maintained old code and newly *re-engineered* code. The other is the productivity, which ranged from eight SLOC/day for new code to 0.2 SLOC/day for old code (40 times worse). Software productivity is actually a very complex product of many factors besides the age of the code [12]. There are many problems with simple productivity factors, but we ignore them to seek a simple approximation and show hypothetical first-order effects based on the age of the code.

We calculate the cost of software code by multiplying the daily labor rate ($50/New SLOC at $100,000 per man year) by the coding productivity (equivalent new SLOC per day). Our simple labor rate allows the readers to easily scale their own labor costs.

ASSUMPTIONS / INPUT DATA		
Labor Cost/MY	$100,000	
Productivity: New SLOC/MY	2,000	(8 SLOC/Day)
Development Cost $/SLOC	50	$/SLOC
SLOC Developed	20,000,000	New SLOC
Total Dev $ (SLOC * Cost/SLOC)	$1,000,000,000	($1B)
Support Factor (40:1 more expensive)	40	
Maint. Software Cost $/SLOC	$2,000	#/SLOC
Life	20	Years
Annual $/Year (10% of Development $)	$100,000,000	
OPTION 1: 100% Maintenance Approach		
Maintained SLOC/Year (Based on Annual $/Software Support Cost)	$100,000,000/ ($2,000 /SLOC) = 50,000	SLOC
% SLOC Maintenance/Year =	50K/20M = 0.25%	
(Clearly you did not get much for the $100M Budget)		
OPTION 2: 100% Effort Is Re-Engineering Each Year		
Maintained SLOC/Year (#SLOC) if 100% Dev. Type Work:	$100,000,000/ ($50/SLOC) = 2,000,000	SLOC
% of Program Enhanced =	10%	

Table 2: *Sample Analysis With Post-Deployment Support Options for a Large Program*

We also do not attempt to quantify considerations in the cost of code since it is beyond the scope of this analysis and well covered by other research [13]. Option 1 in Table 2 shows the results of a typical support project, with the heroic efforts invested in supporting old code. As time goes by and the code ages, this investment will only produce 0.25 percent of *changed* code (50,000 lines) per year due to the poor productivity factor. If all of the investment goes into re-engineering code (i.e. replacing an old module with a brand new one), at the *good* productivity rate of eight SLOC/day, the result would be 2,000,000 source lines being modified as shown in Option 2 of Table 2. Clearly changing 10 percent of the program per year provides a good return on the investment and stays far back from the 15-year rathole age identified by [9].

The equation of interest:

$$\text{New SLOC/Yr} = [B_A * \%RE * (E_{RE} / C_{labor})] + [B_A * \%M * (E_M / C_{labor})]$$

where

B_A = Annual SW Maintenance Budget, in $/year
%RE = Percentage Budget Spent on Re-engineering,
E_{RE} = Efficiency for Re-engineering, (units as SLOC/$)
C_{labor} = Cost of Labor, Assumed Constant
%M = Percentage of Budget Spent on Legacy Maintenance, and
E_M = Efficiency for Maintenance Efforts (Units as SLOC/$)

Note that the amount of code is dependent on the cost per SLOC which is an extremely complex factor. It can even vary inversely to expectations, with newer projects often costing less overall but having a much greater cost per source line of code due to having used compact 4GL languages [12]. Our simple metrics are therefore starting points which allow the reader to make comparisons with their own project or metrics.

Figure 3 (see page 24) shows the results of a sensitivity analysis where we vary the percent of investment into re-engineering and the productivity ratio. The bar heights represent how much total code is changed each year. A proven numeric relationship for the productivity factor as a function of code age would have improved this model but was unavailable.

The desired position to be on this graph is the *back corner* where the team productivity is high (i.e. low ratio) and the percent of code being re-engineered on a planned cycle is high (lots of new functionality possible at lower costs). The result would be newly re-engineered modules that have a lot of the new capa-

bilities that the users want, albeit released at a slow, cyclic pace. Users may be asking to delete functions, add new ones (more code) or just change how the old one works (in theory not adding code).

The product manager or customer liaison team would have to balance the added functionality and deleted features within the annual budget. It would seem prudent however to assume some marginal loss in productivity in determining the amount of code changed each year and allow users to add new capabilities. Hopefully customers and *users* of a software program would be satisfied and *live with* some problems as long as others are getting fixed. Some of the reasons stated for *maintenance* (e.g. security and reliability) are pretty imperative, so it is unlikely that the maintenance can be completely eliminated. Earlier we discussed a target of 25 percent of the annual budget as being necessary for this normal maintenance activity.

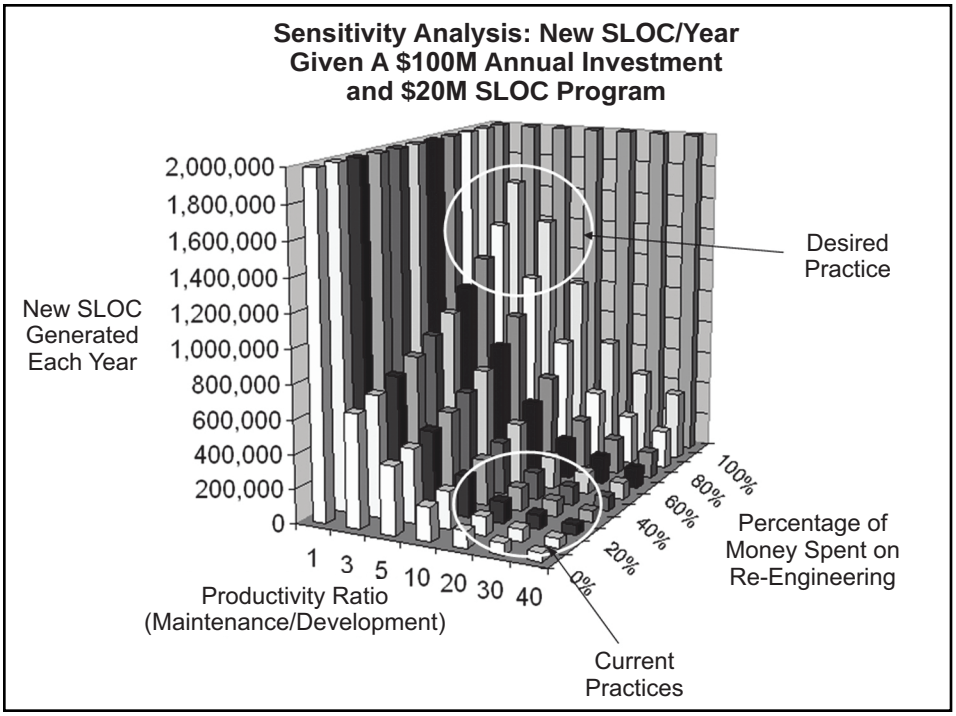
The balance of a *fully funded* annual support budget (that annual 10 cents per initial development dollar) would be 75 percent of the support budget available for *new* functionality via re-engineering. Here we are considering percentages of support dollars however, not percentages of SLOC that comes from including productivity. A 10-year replacement cycle would have an average age of just over five years. It avoids getting into the *rathole* region at 15 years and appears achievable with an allocated mix of 25 percent spent on true maintenance and 75 percent for enhancement via re-engineering.

Some recent research has indicated that the bulk of the maintenance effort gets concentrated on only 20 percent of the program code [14]. The maintenance efforts may indeed focus on a small subset of code, but this does not mean that 80 percent of the code does not ever need re-engineering. Re-engineering provides the users with new functionality and modernizes the code. Thus, managers need to examine systematic replacement of the whole code base in order to keep the average age low and avoid ratholes.

The challenge of improving efficiency in software maintenance is considerable. Our analysis assumed a linear worsening of productivity with age, but in reality productivity rates will need to be aggressively managed to be successful. There are several strategies that could be used to do this, namely the following:

- **Use modern tools.** We have shown numerically that cyclic replacement and minimized average code age can have positive effects. Most large, old programs have both old and new modules with ready candidates for replacement, but even newly completed projects must start aggressively replacing modules since a permanent bias will creep into the average age for each year of delay.
- **Modular software design.** This is the basis of object-oriented design principles, and benefits the code maintainer by allowing easier replacement of modules. The method called software *re-factoring* advocates replacing software on a modular basis as developed by Ward Cunningham [15] and Martin Fowler [16]. A modular strategy may help maintenance by providing defined interfaces and functionality.
- **Add more off-the-shelf code.** This leverage of existing code or products can save support cost, but this can also cause problems. Some vendors change their product or stop supporting a version, thus increasing maintenance costs. Databases, publish/subscribe services, and web servers and browsers are all modern examples of products that can modernize code.
- **Staff rotation.** Regular rotation of staff between development and maintenance teams could minimize the differences in skill sets, productivity, experiences, and employee enthusiasm. Engineers that have maintenance experience know first-hand why it is important to develop code for maintenance.
- **Modern languages.** New software languages that use fewer lines of code to do the same function can help lower support costs. Alan Albrecht at IBM developed methods to use function points as a measure of software functions to estimate program size [17]. Current data shows that HTML and Web-service methods take 25 percent less SLOC than C+ or Java for similar function points (functionality) [18]. Another study [19] showed that a representative program of 300 Web objects (function points, links, multimedia files, scripts and Web building blocks) took 38 percent fewer person-months to develop when HTML was used instead of Java. This implies that added emphasis should be placed on replacing older code with newer software structures and languages in order to capture the improved productivity rate.

Figure 3: *The Amount of Code Changed as a Function of the Mix of Maintenance Investment and Productivity Factors*



Concluding Remarks

Software support is expensive, is absolutely necessary, and will be necessary for many years. In military programs, the support costs are borne by the taxpayers, and in commercial projects, support expense is subtracted from sales or licensing revenue. Strategies for cyclic recapitalization of the code to keep the average age young and improve programming productivity should minimize long-term maintenance costs and allow the support team to regularly add improve-

ments and functionality.◆

References

1. Boehm, Barry W. Software Engineering Economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.
2. United States Air Force. Software Technology Support Center. Guidelines for Successful Acquisition of Software Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems (GSAM). Version 3.0 May 2000 <www.stsc.hill.af.mil/resources/tech_docs/>.
3. Piersall, COL James. "The Importance of Software Support to Army Readiness." Army Research, Development, and Acquisition Bulletin. Jan.-Feb. 1994.
4. Pressman, Roger S. Software Engineering: A Practitioner's Approach 3rd ed. New York, NY: McGraw-Hill, 1992.
5. Glass, Robert L. Building Quality Software. Englewood Cliffs, NJ: Prentice Hall, 1992.
6. Banker, Rajiv, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software Errors and Software Maintenance Management. The Netherlands: Kluwer Academic Publishers, 2002.
7. Lehman, M.M. "Programs, Life Cycles, and Laws of Software Evolution." Proc. of the IEEE 68.9 (Sept. 1980): 1060-1076.
8. Eisenbach, Susan. "Software Maintenance, No Jokes, Lectures 2 and 3." <www.doc.ic.ac.uk/~sue/475/lec%202%202%20&%203%20-%20maintenance.pdf>.
9. Lano, Kevin and Howard Haughton. Reverse Engineering and Software Maintenance. New York: McGraw Hill, 1994.
10. Wade, Stu and Andy Laws. Legacy System Management via the Triage Model, Software Triage. Liverpool, UK: John Moores University, 1998 <www.cms.livjm.ac.uk/research/docs/CMS1898.DOC>.
11. Perry, William E. "Don't Pour Money Down Rat Holes that Infest Your Budget." Government Computing News Dec. 1993.
12. Jones, Capers. Programming Productivity. New York: McGraw Hill, 1986.
13. Hinh, Jarus M. "The Impact of Faster, Better, Cheaper." Spacecraft Ground Systems Architecture Workshop, 1999 <http://sunset.usc.edu/events/GSAW/gsaw99/dpdf-presentations/breakout-2/hihn-1.pdf>.
14. Boehm, Barry W. "The Economics of Software Reliability." International Symposium on Software Reliability Engineering, 19 Nov. 2003 <www.cs.colostate.edu/~malaiya/issre/barryboehm.pdf> .
15. Beck, Kent. eXtreme Programming eXplained: Embrace Change. Reading, MA: Addison Wesley Longman, 1999.
16. Fowler, Martin, et. al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
17. Albrecht, Alan J. "Measuring Application Development Productivity." Proc. of the Joint SHARE/GUIDE/IBM Application Development Symposium, Oct. 1979.
18. Quantitative Software Management, Inc. "Function Point Programming Languages Table." Version 2.0 2002 <www.qsm.com/FPGearing. html>.
19. Reifer, Donald J. "Estimating Web Development Costs: There Are Differences." CROSSTALK June 2002 <www.stsc.hill.af.mil/crosstalk/2002/06/reifer.html>.

About the Author

David Lechner currently works as a consultant for GeoLogics Corp., and recently founded WeC2 Technologies, a small business focusing on Web-enabled C2 software. He has 20 years of experience in Department of Defense systems development and acquisition. Lechner's civil service career includes: Naval Air Systems Command (Reconnaissance, Electronic Warfare, Special Operations, Navy); Naval Electronic Systems Command (PMW143); General Systems Administration (Federal Systems Integration and Management Center); and Naval Sea Systems Command (PMS425). His private industry experience includes DRS Electronics Systems and GeoLogics Corp. Lechner has a bachelor of science in Electrical Engineering from Carnegie Mellon University, a M.E.Ad. from George Washington University, and a master of science in Computational Physics from George Mason University.

GeoLogics Corp.
5285 Shawnee RD
STE 300
Alexandria, VA 22312
Phone: (240) 418-1544
Fax: (703) 750-4010

COMING EVENTS

November 30-December 8
QFD 2006
18th Symposium on QFD
Austin, TX
www.qfdi.org/symposium.htm

December 3-8
LISA '06
20th Large Installation System Administration Conference
Washington, D.C.
www.usenix.org/events/lisa06

December 4-7
ICSOC 2006
4th International Conference on Service Oriented Computing
Chicago, IL
www.icsoc.org/

December 4-7
IITSEC 2006 Interservice/Industry Training Simulation and Education Conference
Orlando, FL
www.iitsec.org/index.cfm

December 5-7
XML 2006
Boston, MA
http://2006.xmlconference.org

December 11-15
ACSAC 2006 Annual Computer Security Applications Conference
Miami, FL
www.acsac.org

December 14-15
11th Annual ITC East 2006 Conference
Harrisburg, PA
www.govresources.com/itceast2006.html

June 18-21, 2007
2007 Systems and Software Technology Conference
Tampa, FL
www.sstc-online.org

Earned Schedule: An Emerging Enhancement to Earned Value Management

Walt Lipke
Retired Deputy Chief Software Division, Oklahoma City Air Logistics Center

Kym Henderson
Education Director, Project Management Institute, Sydney Chapter

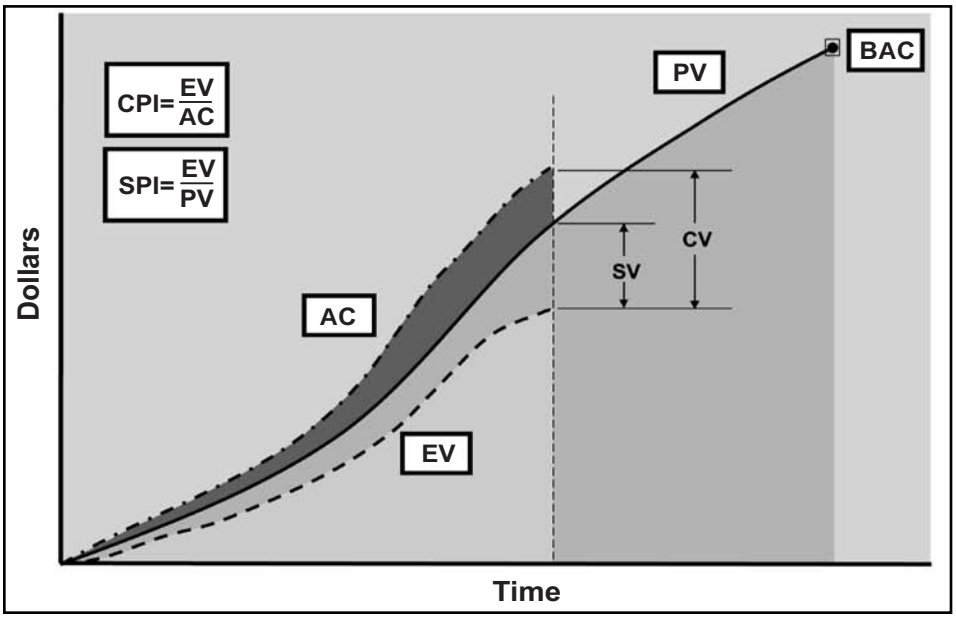
Earned Schedule (ES) is a method of extracting schedule information from Earned Value Management (EVM) data. The method has been shown to provide reliable schedule indicators and predictors for both early and late finish projects. ES is considered a breakthrough technique to integrated performance management and EVM theory and practice. The method has propagated rapidly and is known to be used as a management tool for software, construction, commercial, and defense projects in several countries, including the United States, Australia, United Kingdom, Belgium, and Sweden. The principles of ES have been included in the "Project Management Institute College of Performance Management, Practice Standard for EVM" as an emerging practice [1].

EVM was created within the U.S. Department of Defense in the 1960s and has shown over the four decades from that time to be a very valuable project management and control system. EVM uniquely connects cost, schedule, and requirements thereby allowing for the creation of numerical project performance indicators. Managers now have the capability to express the cost and technical performance of their project in an integrated and understandable way to employees, superiors, and customers.

For all of the accomplishments of EVM in expressing and analyzing cost performance, it has not been as successful for schedule performance. The EVM schedule indicators are, contrary to expectation, reported in units of cost rather than time. And, because cost is the unit of measure, the schedule indicators require a period of familiarization before EVM users and project stakeholders become comfortable with them and their use. Beyond this problem, there is the much more serious issue: The EVM schedule indicators fail for projects executing beyond the planned completion date.

Because these problems are well known to EVM practitioners, over time the application has evolved to become a management method focused primarily on cost. The schedule indicators are available, but are not relied upon to the same extent as the indicators for cost. The resultant project management impact from the EVM schedule indicator issues is cost and schedule analyses of project status and performance have become disconnected. Cost analysts view the EVM cost reports and indicators while schedulers tediously update and analyze the network schedule. Frequently for large projects, these separate skills are segregated and, often, their respective analyses are not reconciled.

Figure 1: *Earned Value*



It has been a long expressed desire by EVM practitioners to have the ability to perform schedule analysis from EVM data similarly to the manner for cost. Various approaches to using earned value (EV) for this analysis have been proposed and studied from time to time. However, none of the methods have proven to be satisfactory for both early and late finishing projects.

Before discussing the ES approach to overcoming the described cost-schedule dilemma, let us first review EVM.

EVM Measures and Indicators

EVM has three measures: planned value (PV), actual cost (AC), and EV. Refer to Figure 1 as an aid to this discussion. The planned values of the tasks comprising the project are summed for the periodic times (e.g., weekly or monthly) chosen to status project performance. The time-phased representation of the planned value is the performance management baseline (PMB). AC and EV are accrued and are likewise associated with the reporting periods. For each measure, the time-phased graphs are characteristically seen to be S-curves. Observe that PV concludes at the Budget at Completion (BAC), the planned cost for the project. The BAC is the total amount of PV to be earned.

From the three measures, project performance indicators are formed. The cost variance (CV) and cost performance index (CPI) are created from the EV and AC measures, as follows: CV = EV - AC and CPI = EV/AC. In a similar manner, the schedule indicators are: SV = EV - PV, and SPI = EV/PV, where SV is the schedule variance and SPI is the schedule performance index.

Now examine the formulation of the schedule indicators and recall that the PV

and EV curves conclude at the same value, BAC. The fact that PV equals BAC at the planned completion point and does not change when a project runs late causes the schedule indicators to falsely portray actual performance. In fact, it is commonly observed that the schedule indicators begin this behavior when the project is approximately 65 percent complete.

The irregular behavior of the schedule indicators causes problems for project managers. At some point it becomes obvious when the SV and SPI indicators have lost their management value. But, there is a preceding gray area, when the manager cannot be sure of whether or not he should believe the indicator and subsequently react to it. From this time of uncertainty until project completion, the manager cannot rely on the schedule indicators portion of EVM.

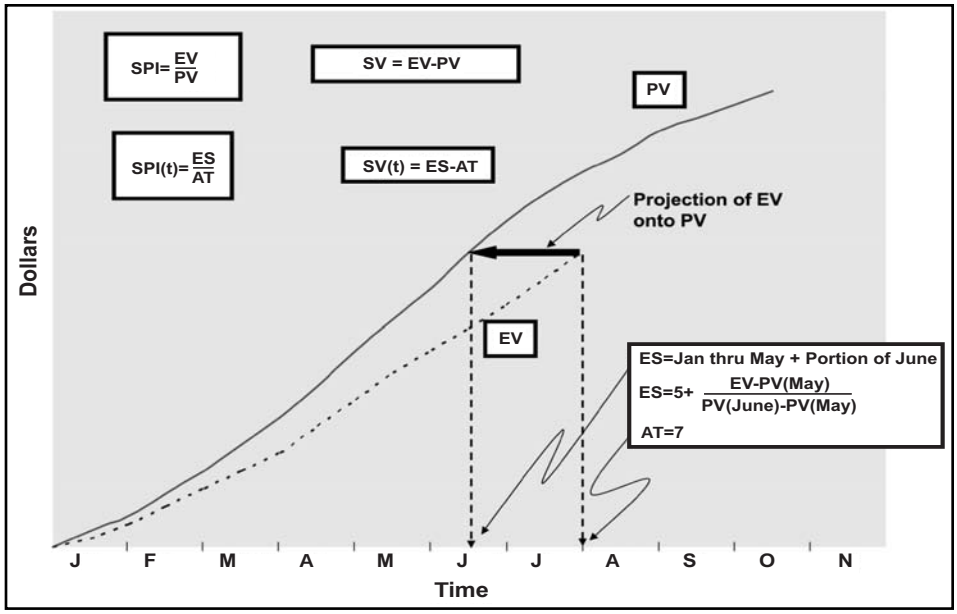


Figure 2: *ES Concept*

Earned Schedule Description

The technique to resolve the problem of the EVM schedule indicators is ES. The ES idea is simple: Identify the time at which the amount of EV accrued should have been earned [2]. By determining this time, time-based indicators can be formed to provide schedule variance and performance efficiency management information.

Figure 2 illustrates how the ES measure is obtained. Projecting the cumulative EV onto the PV curve (i.e., the PMB), as shown by the diagram, determines where PV equals the EV accrued. This intersection point identifies the time that amount of EV should have been earned in accordance with the schedule. The vertical line from the point on the PMB to the time axis determines the *earned* portion of the schedule. The duration from the beginning of the project to the intersection of the time axis is the amount of ES.

With ES determined, it is now possible to compare where the project is timewise with where it should be in accordance with the PMB. *Actual time*, denoted AT, is the duration at which the EV accrued is recorded. The time-based indicators are easily constructed from the two measures, ES and AT. SV becomes SV(t) = ES - AT, and SPI is SPI(t) = ES/AT.

The graphic and the box in the lower right portion of Figure 2 portray how ES is calculated. While ES could be determined graphically as described previously, the concept becomes much more useful when facilitated as a calculation. As observed from Figure 2, all of the PV through May has been earned. However, only a portion of June has been completed with respect to the baseline. Thus the duration of the completed portion of the planned schedule is in excess of five months. The EV accrued appears at the end of July, making actual time equal to seven months. The method of calculation to determine the portion of June to credit to ES is a linear interpolation. The amount of EV extending past the cumulative PV for May divided by the incremental amount of PV planned for June determines the fraction of the June schedule that has been earned.

Evolution of Earned Schedule

The ES concept was conceived during the summer of 2002 and was publicly introduced in March 2003 with *The Measurable News* article, *Schedule Is Different* [2]. This was quickly followed a few months later by the complementary article, *Earned Schedule: A Breakthrough Extension to Earned Value Theory? A Retrospective Analysis of Real Project Data* [3]. Using EVM data from several completed real projects, this second article verified the ES measure and its derivative indicators functioned as described in the seminal article *Schedule Is Different*. From that time, the behavior of the calculated measure of ES and its indicators has been verified many times by practitioners using real data from various types of projects.

Schedule Is Different alluded to the potential of using ES to forecast when a project would complete, but did not develop the equations. The second article [3] identified a schedule duration predictor analogous to the predictor for final cost, BAC/CPI. This schedule predictor, PD/SPI(t), where PD is the planned duration, was applied to real data and demonstrated the potential of the project duration and completion date prediction using ES.

Following the second article was *Further Developments in Earned Schedule* [4]. This article further expanded the ES schedule prediction and algebraically compared the ES methods with other published techniques. Two ES predictive calculation methods were identified as the *short form* and *long form*. The short form is as described previously, IEAC(t) = PD / SPI(t), where IEAC(t) is termed the Independent Estimate at Completion (time). The long form, just as for the short form, mimics an equation for forecasting final cost: IEAC = AC + (BAC - EV) / PF, where PF is a selected performance factor [1]. The long form schedule duration equation is as follows: IEAC(t) = AT + (PD - ES)/ PF(t), where AT is the actual duration, and PF(t) is a selected time performance factor.

In the *Further Developments in Earned Schedule* article, two common methods of schedule prediction were used for comparison to the predictive performance of ES [5]. The first method uses SPI from EVM, and the second applies a performance factor termed the *critical ratio*. The critical ratio is equal to SPI multiplied by CPI. The short form results were compared against two scenarios, early finish and late finish performance. Using data from two real projects discussed in the article, the results for the three forecasting methods are tabulated in Table 1 (see page 28) [4]. Only the ES forecast yielded correct results for both early and late

completion. Neither of the other two methods provided correct results in either scenario.

In the same article, the long form equation was shown to provide correct end point results, regardless of the PF(t) used [4]. Thus, the long form equation possesses the identical characteristic of its companion equation for forecasting final cost. This characteristic of calculating and obtaining the correct result at project completion is required for the exploration and research of potential schedule based performance factors.

As the application of ES grew, it was recognized that there needed to be a common set of terminology. The interested parties involved agreed to a common theme: The terms should be parallel to, but readily distinguishable from those of EVM. It was thought that these characteristics would encourage the application of ES by minimizing the learning curve required. As seen from Table 2, ES Terminology, the chosen terms are comparable to those from EVM. In most instances, the ES term is simply the analogous EVM term appended by the suffix (t).

After the ES method was published in March 2003, it rapidly became viewed as a viable extension to EVM practice. By fall of 2003, the Project Management Institute - College of Performance Management (PMI-CPM) had become interested in the new practice. Within the next year an *emerging practice* insert citing the principles of ES was included in the 2004 release of the PMI-CPM *Practice Standard for EVM* [1].

With increasing use and interest in ES came the question, *does ES provide the long sought bridge between EVM and the network schedule?* Mainstream EVM thought is that other than the creation of the PMB, there can never be a strong connection between these two management components. The reasoning is EVM provides a macro-type assessment of performance but cannot yield the detail required to assess the true schedule performance.

Two articles, one published June 2005 and the other spring 2005, addressed the question of how ES contributes to making the direct connection between the schedule and the EVM data. The June 2005 article is appropriately titled *Connecting Earned Value to the Schedule*, while the spring 2005 article is *Earned Schedule in Action* [6, 7]. The *Connecting Earned Value* article describes how ES facilitates the bridge. The value of ES coincides with a PV point on the PMB. In turn, the PV is directly connected to specific tasks or work packages either completed or in work. Having this identification allows determination of how well the schedule is being followed. Differences in plan versus the actual distribution of EV provide insight as to which tasks may have impediments constraining progress and which have the possibility of future rework. The article introduces a measure of schedule adherence, directly connecting EVM to the network schedule, termed the *P-Factor* [6]. This new measure has lead to a theory which may prove to yield earlier and better prediction for both cost and schedule.

A considerable amount of interest has been shown for *Earned Schedule in Action*. The article compares the results from applying ES and Critical Path (CP) duration prediction methods to a small scale yet time-critical IT project. What was observed during project execution is the duration predicted from ES converged to the actual, final value from the pessimistic side, while the forecast from CP analysis converged optimistically. Because the ES predictive method takes into account past schedule performance while the CP method may not, it has been conjectured that, in general, ES yields a more consistently reliable schedule forecast. Further research is needed to confirm this hypothesis.

One advantage of ES became obvious in the CP study. Prediction obtained from ES calculations is considerably less effort than the CP approach, which requires very detailed task-level bottom-up analysis of the network schedule.

As a final point, the two articles discussed here provide rationale for the position that ES *bridges* the two disciplines of EVM and network schedule analysis. Even so, just as for cost, neither EVM nor ES can completely supplant bottom-up estimation techniques. For both, their respective predictive calculations are useful as macro methods for rapidly generating estimates and as a cross check of the corresponding bottom-up analysis.

Table 1: *IEAC(t) Comparison*

	Early Finish Weeks	**Late Finish** Weeks
Planned Duration	25	20
Actual Duration	22	34
CPI	2.08	0.52
SPI	1.17	1.00
SPI(t)	1.14	0.59
PD/SPI(t)	22.0	34.0
PD/SPI	21.4	20.0
PD/(CPI * SPI)	10.3	38.7

Applications

Early in the existence of ES, some construed that the methods are limited in application. They believed that ES could only be used successfully for small information technology (IT) type projects. This perception occurred because software and IT projects were the environments in which the concept was created and first applied. The presumption is demonstrably false. ES is scalable up or down, just as is EVM. As well, ES is applicable to all types of projects, as is EVM. It follows that the scalability and applicability characteristics must exist; *after all, ES is derived from EVM.*

ES is known to be used in several organizations and countries for a variety of project types. Small IT and construction projects as well as large defense and commercial endeavors have employed and continue to include ES as part of their management toolset. The users have reported an increased ability to forecast future outcomes and the capability to identify late occurring problems that are masked when viewing EVM data alone. Significant applications in the

United States are at Lockheed Martin, Boeing Dreamliner, and the Air Force (use in acquisition oversight). The United Kingdom Ministry of Defence has identified two major programs applying ES: Nimrod (maritime patrol aircraft) and Type 45 (Naval destroyer). Several smaller applications, mostly IT-related projects, have occurred in Belgium by Fabricom Airport Systems, as well as in the United States and Australia.

Research

Small-scale research has occurred throughout the evolution of ES. Each idea and next step has been applied and examined against real project data. However, due to data limitations, the testing and conclusions are not considered sufficiently complete. Although lack of testing is a drawback, the risk associated with ES usage is minimal. One compelling point supporting ES is that, regardless of the circumstances of the application (who, project type, company, country), the findings from all sources are consistent. The ES method, in every application, outperforms other EVM-based methods for representing schedule performance.

A research team at the University of Ghent, Belgium has recently published findings comparing ES to other project duration methods based on EVM measures [8]. Their conclusions coincide with the statement above; ES is the better performer. This research team has aspirations to perform rigorous testing of ES and the other prediction methods, using simulation techniques. They have also indicated interest in exploring the implications of the P-Factor (the measure of schedule adherence) discussed earlier.

What's Next?

The expectation is the application of ES will continue to expand and propagate, coincident with the worldwide expansion of EVM. As ES is used more and more, it is reasonable to believe there will be increasing demand for its inclusion in EVM tools. Our conjecture is that the availability of tools employing ES is forthcoming in the near future. Along with increased application and tool availability, ES training will be requested as part of the provided EVM course. And most certainly as the use of ES expands, more information will be published, which will improve and mature the method and add to a rapidly expanding *ES Body of Knowledge*. Ultimately, we foresee that ES will become generally accepted and subsequently included within Earned Value Management standards and guidance. Finally, it is our belief that ES will lead to improved prediction techniques for both cost and schedule.

	EVM	Earned Schedule
Status	Earned Value (EV)	Earned Schedule (ES)
	Actual Costs (AC)	Actual Time(AT)
	SV	SV(t)
	SPI	SPI(t)
Future Work	Budgeted Cost for Work Remaining (BCWR)	Planned Duration for Work Remaining (PDWR)
	Estimate to Complete (ETC)	Estimate to Complete (time) ETC(t)
Prediction	Variance at Completion (VAC)	Variance at Completion (time) VAC(t)
	Estimate at Completion (EAC) (supplier)	Estimate at Completion (time) EAC(t) (supplier)
	Independent EAC (EAC) (customer)	Independent EAC (time) IEAC(t) (customer)
	To Complete Performance Index (TCPI)	To Complete Schedule Performance Index (TSPI)

Table 2: *ES Terminology*

Available Resources

There is a considerable amount of accessible ES information to aid current and potential users. Published papers, conference presentations, and workshop materials are available from two Web sites: <www.earnedschedule.com> and <http://sydney.pmichapters-australia.org.au/> (Education, then Papers and Presentations). Both sites offer downloading of the information free of charge. Additionally, calculators facilitating the application of ES are available from <www.earnedschedule.com>.

Summary

ES was created as a non-complex solution to resolve the problem of the EVM schedule indicators failing for late-finishing projects. The ES method requires only the data available from EVM and has been shown to provide better prediction than other EVM-based methods. Duration forecasting using ES is easier to do than detailed, bottoms-up estimation, and possibly yields better results, as well. ES is scalable up or down, and it is applicable to any project using EVM. ES facilitates identification of tasks with possible impediments, constraints, or future rework and has the potential to improve both cost and schedule prediction.

ES is a powerful new dimension to integrated project performance management and practice. It has truly become a breakthrough in theory and application.◆

References

1. Project Management Institute. Practice Standard for EVM. PMI, 2004.
2. Lipke, Walt. "Schedule Is Different." The Measurable News Mar. 2003: 10-15.
3. Henderson, Kym. "Earned Schedule: A Breakthrough Extension to Earned Value Theory? A Retrospective Analysis of Real Project Data." The Measurable News Summer 2003: 13-23.
4. Henderson, Kym. "Further Developments in Earned Schedule." The Measurable News Spring 2004: 15-22.
5. Anbari, Frank T. "Earned Value Project Management Method and Extensions." Project Management Journal 34.4 (Dec. 2003): 12-23.
6. Lipke, Walt. "Connecting Earned Value to the Schedule." CROSSTALK June 2005 <www.stsc.hill.af.mil/crosstalk/2005/06/0506Lipke.html>.
7. Henderson, Kym. "Earned Schedule in Action." The Measurable News Spring 2005: 23-30.
8. Vanhoucke, Mario, and Stephan Vandevoorde. "A Comparison of Different Project Duration Forecasting Methods Using Earned Value Metrics." International Journal of Project Management 24.4 (May 2006): 289-302.

About the Authors

Walt Lipke recently retired as the deputy chief of the Software Division at the Oklahoma City Air Logistics Center. He has more than 35 years of experience in the development, maintenance, and management of software for automated testing of avionics. During his tenure, the division achieved several software process improvement milestones, including first Air Force activity to achieve Level 2 of the Software Engineering Institute's Capability Maturity Model (CMM) in 1993; the first software activity in federal service to achieve CMM Level 4 distinction in 1996; division achieved ISO 9001/TickIT registration in 1998; and the division received the SEI/IEEE Award for Software Process Achievement in 1999. He is the creator of Earned Schedule©, which extracts schedule information from earned value data. Lipke is a graduate of the U.S. Department of Defense course for Program Managers. He is a professional engineer with a master's degree in physics.

1601 Pembroke DR
Norman, OK 73072
Phone: (405) 364-1594
E-mail: waltlipke@cox.net

Kym Henderson is currently the education director of the PMI's Sydney Australia Chapter. His IT career has covered project and program management, software quality assurance management, and project planning and control. Henderson has worked for a number of companies in many industry sectors including commercial, defense, government, manufacturing, and telecommunications and financial services. He has extensive experience in project recovery, where the use of simplified EVM techniques to assist in rapidly evaluating current project status has proven invaluable. Henderson has received several awards, including a Reserve Force Decoration for 15 years' efficient service as a commissioned officer in the Australian Army Reserve. He has a master of science in computing from the University of Technology, Sydney.

P.O. Box 687
Randwick NSW, 2031
Sydney, Australia
Phone: +61 414 428 537
Fax: +61 283 949 295
E-mail: kym.henderson @froggy.com.au

WEB SITES

Jim Collins

www.jimcollins.com/lib/articles.html#

We must reject the idea that the primary path to greatness in the social sectors is to become more like a business. Most businesses fall somewhere between mediocre and good. Few are great. When you compare great companies with good ones, many widely practiced business norms turn out to correlate with mediocrity, not greatness. So, then, why would we want to import the practices of mediocrity into the social sectors?

Agile Alliance

www.agilealliance.org

The Agile Alliance (AA) is a non-profit organization that supports individuals and organizations who use Agile approaches to develop software. Driven by the simple priorities articulated in the Manifesto for Agile Software Development, Agile development approaches deliver value to organizations and end users faster and with higher quality. The AA exists to help more Agile projects succeed and to help the enthusiasts start more Agile projects. The AA is uncovering better ways of developing software by doing it and helping others do it, and have come to value individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.

Déjà Review

This month CROSSTALK echoes the clarion call back to basics. Been there, done that? Not so fast; while the bleeding edge of technology can be exciting, every engineer understands the importance of a good technical foundation.

Confucius reminds us: "Study the past if you would define the future." [1]

In our case (engineers), study the essential ingredients, principles, and procedures learned in our formative years that form the heart of our profession. A technical déjà vu or déjà review.

This summer, I experienced an academic déjà vu. My son decided to start his college career early by enrolling in a chemistry class at the BCS[2]-busting University of Utah. Chemistry in the summer? I give him credit – the summer of my high school senior year social bonds trumped covalent bonds.

Nevertheless, I accompanied him to his first class. His mother instructed him to sit in the front of the class. I corrected him by offering him the advantages of a good seat in the back of the lecture hall: 1) a quick exit to the next class; 2) easy to step out for a snack, and 3) a good buffer from the explosions, recalling most chemistry professors are closet pyromaniacs.

We settled into our seats. Why are college seats smaller these days? In typical academic fashion, the professor started in the middle of a lecture with no set-up, explanation, or introduction. Bam! He is rambling on about Sig Figs. Although the term was familiar, to be honest, I was not sure if he was promoting tobacco products, a local fraternity, or soft chewy cookies rumored to be named after Sir Isaac Newton.

Gaining my bearings, I ascertained the lecture to be a primer on Significant Figures (Sig Figs), a basic concept used in chemistry, physics, engineering, and disciplines that rely on measurements. It was a nice refresher. The following are sample questions used in the lecture. They are simple and you are a professional, so have a go at them.

Identify the number of significant figures:

a) 0.00072
b) 2.07200
c) 500
d) 210.0

Answer according to the rules of significant figures:

e) 4.7832 + 1.234 + 2.02
f) 1.0236 – 0.97268
g) 2.8723 x 1.6
h) 45.2 ÷ 6.3578

Write down your answers. Before checking your answers, review the following Sig Fig rules:

- *Non-zero digits are always significant.*
- *Leading zeros are never significant.*
- *Zeroes between two significant digits are significant.*
- *Trailing zeros are significant only if the decimal point is specified.*
- *For addition and subtraction, the last digit retained is set by the first doubtful digit.*
- *For multiplication and division, the answer contains no more significant figures than the least accurately known number.*

Review the questions and your answers and make any corrections based on the Sig Fig rules. Now check your answers against the answers below[3].

How did you fare initially? How did you fare after a quick review of the rules? Did your answers improve or was there little to improve upon? For those still confused a few examples of applying each rule would bring you onboard. That is the power of technical déjà review.

The Sig Fig rules provide a veiled insight on revitalizing and reinforcing your technical foundation.

Engineering is an amalgamation of several disciplines – mathematics, physics, chemistry, and electronics to name a few. Like non-zero digits, foundational disciplines are always significant in an engineer's performance.

Constantly exposed to unproven theories, products and solutions, engineers should dismiss unfounded approaches like a scientist disregards leading and trailing zeros. Stick with proven, verified, and documented technologies. There is a reason it is called the bleeding edge.

Just as zeroes between significant digits are significant, an engineer's true value lies in understanding the interrelationships, limitations, and synergy between various system technologies and the foundational disciplines they are based upon.

Engineers have strengths and weaknesses. Strong skills lead to specialization while pride downplays weaknesses. In the vein of the Sig Fig rule for addition and subtraction, an engineer is only as effective as his most doubtful skill, and akin to the Sig Fig rule for multiplication and division, an engineering team contains no more effectiveness than the least accurate member. Hence, surround yourself with colleagues who complement your strengths and offset your weaknesses.

Do not let your engineering/management skills rust. Map out your own déjà review, revitalize your technical skills and lay a sound foundation for your career.

— Gary A. Petersen
Shim Enterprises, Inc.
gary.petersen@shiminc.com

Notes

1. Confucius, a Chinese philosopher and reformer (551-479 B.C.).
2. Bowl Championship Series, a system that selects the college football matchups for five prestigious bowl games.
3. Answers: a) two; b) six; c) one; d) four; e) 8.04; f) 0.0509; g) 4.6; h) 7.11.

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 perspective 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>.

CrossTalk is co-sponsored by the following organizations:

76th SOFTWARE MAINTENANCE GROUP

309th SOFTWARE MAINTENANCE GROUP

402d SOFTWARE MAINTENANCE GROUP

Homeland Security

NAVAIR

CrossTalk / 517 SMXS/MXDEA
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

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