Software Knowledge Management
Strengthening Our Community of Practice

Lt. Col. Joe Jarzombek
ESIP Director

When we do something together over time, we create shared practices. We learn to do what needs to be done, we learn about each other, and we develop shared ways of doing things. We form *communities of practice* in which sharing is a natural part of belonging. Indeed, "knowing" implies social communities, because facts alone have no meaning outside a shared context.

Our software community of practice is such a familiar experience that many people may hardly notice it. Its identity derives from participation, and it is self-organized around important matters through dynamic renegotiation of what the profession is about and what needs doing and learning. The boundaries are defined by actual participation, not by affiliation or title. It follows the contours of common practice and is held together by knowledge rather than task. Interaction among practitioners continually refines and develops the elements of the practice. That which is an improvement is adopted into the body of knowledge. Failure to interact regularly with other members of the community will eventually result in estrangement from it. These realities are indicative of why the identity of our software community of practice is continuing to evolve.

Our software community of practice develops resources such as shared learning and practices. Organizations such as the Software Engineering Institute (SEI) and the Software Technology Support Center (STSC) facilitate the capture and transfer of knowledge critical to our software community practitioners. Through STSC-refereed articles, *CrossTalk* functions as one of our software community's key conduits for transferring knowledge. Important on-line resources have also become part of many organizations' virtual infrastructure, particularly the Web sites of the SEI,[1] the Software Engineering Information Repository,[2] the STSC,[3] the Embedded Computer Resources Support Improvement Program (ESIP),[4] the Data and Analysis Center for Software,[5] the Software Program Managers Network,[6] and the Defense Acquisition Deskbook.[7]

The annual Software Technology Conference[8] and SEI Symposium and Software Engineering Process Group Conferences[9] are forums that provide vital interaction opportunities for our software community because they provide the necessary facilities of belonging: alignment, engagement, and exploration. They provide time for reflection and the unstructured personal contact so vital in the exchange of information and the development of community resources. As valuable as the seminars are in each of these forums, I contend that in their absence, many people would continue to pay to attend the conferences for the networking opportunities alone. Perhaps, to extend and energize our community of practice, we should consider offering a new conference registration category called "networking only."

An organization's software community of practice is critical to its success. The community may exist informally within and across business units and projects and often across organizational boundaries. To gain the most leverage, it maintains links outside the organization to strengthen its knowledge base. Communities of practice are organizational assets because of the knowledge they steward at their core and through the learning they inspire at their boundaries. The learning potential of an organization resides in the interaction of cores and boundaries in "constellations" or clusters of different communities of practices.

I contend that the use of the Capability Maturity Model Integration (CMMI)[10] product suite, being released this year, will have one of the more revolutionary impacts on organizational constellations of communities of practice because the CMMI addresses enterprise-wide, integrated process improvement that cuts across traditional disciplinary boundaries. What will transform an organization or community of practice is not what an individual knows or single function controls but what a group knows and causes to happen. Processes and practices that cross disciplines and functions provide the basis for group "passion" or motivation. Use of integrated knowledge within and among communities of practice should prove to be the most sustainable and profitable aspect of any organization.

Organizations cannot truly manage knowledge because it is tacit or internal to individuals; however, they can manage the environment necessary for the community of practice to flourish and share information that is a product of that knowledge. For organizations to successfully compete in an era of rapid change, they need to invest in connectivity more than information. Using the leveraging capabilities of the Internet and Intranets, organizations need to establish their own virtual knowledge management infrastructure that evolves

see COMMUNITY, page 10

On the front cover:

From the upper right corner, clockwise: Kevin Tjoland (TISFD), David Haakenson (TISFB), Ken Raisor (TISHD [TaskView TSP Project]), Mark Peterson (TISFD), David Webb (TISHD [TaskView TSP Project]).

Using the TSP on the TaskView Project

David Webb, *Ogden Air Logistics Center, Software Engineering Division*
Watts S. Humphrey, *Software Engineering Institute*

This article reports the first results of using the Team Software Process (TSP)TM on a software-intensive system project. The TSP was developed by the Software Engineering Institute (SEI) to guide integrated teams in producing quality systems on their planned schedules and for their committed costs. The TaskView team at Hill Air Force Base, Utah used the TSP to deliver the product a month ahead of its originally committed date for nearly the planned costs. Because the engineers' productivity was 123 percent higher than on their prior project, they included substantially more function than originally committed. Testing was completed in one-eighth the normal time, and as of this writing, the customer has reported no acceptance test defects.

This article describes the experiences of a team that used the TSP to produce a software-intensive product for the U.S. Air Force. The Ogden Air Logistics Center, Software Engineering Division, Hill Air Force Base, Utah, has a long history of producing avionics and support software for the Air Force. The division had previously been assessed at a Capability Maturity Model (CMM)® Level 3 and has just recently been assessed at CMM Level 5. TaskView, one of the products they delivered, is a system to help Air Force pilots produce flight plans. Flight planning is labor-intensive and time-consuming; TaskView automates much of this work. It helps mission planners produce accurate flight plans with less labor and in less time than previously possible. The project was completed ahead of its original schedule and within its committed budget. The product is currently in customer acceptance testing with no defects reported to date. This article is the first published report of project results with the TSP.

Following a brief TSP overview, we describe the software organization, the TaskView project, and the team's experiences in introducing and using the TSP. Next, we cover the engineers' reactions to using this process. We conclude with a brief summary of the key findings from the TaskView experience. The division already had a high-maturity software process, so it had data available from prior work. We can thus compare the performance of the TSP team to previous projects. Although this article presents some of the data, we only show a few of the indicators that are potentially available for TSP projects.

The SEI's work is supported by the Department of Defense.

Personal Software Process, PSP, Team Software Process, and TSP are service marks of Carnegie Mellon University. Capability Maturity Model and CMM are registered trademarks of Carnegie Mellon University.

The TSP

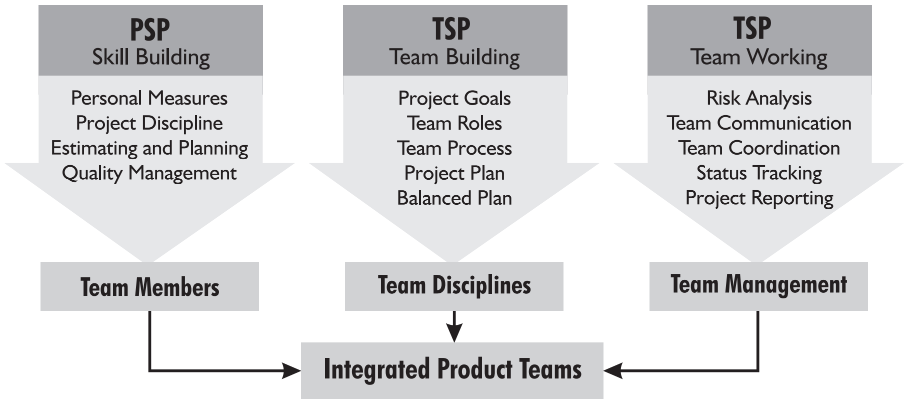
Although the concepts and methods for running integrated teams are well known, the specific steps often are not obvious to working engineers and managers. For example, to be effective, teams need precise goals, clearly stated roles, a defined engineering process, and a detailed plan for the work. They need a framework for periodic coordination and structured methods to review and track project risks and issues. Team measures must be defined and recorded, tracking mechanisms developed, and a reporting system established.

Although none of these items is particularly complex or difficult, the specific actions often are not obvious. Before engineers can work effectively in an integrated team environment, they need to know precisely what to do. If they have not done such work before or do not have a detailed process to guide them, they will generally defer the new or unfamiliar items until they know how to handle them. They then do the tasks they fully understand. As a result, many of the actions required for effective teaming do not get done. Teams can waste a great deal of time trying to establish goals, resolving their working relationships, and figuring out how to do the work.

How the TSP Works

The TSP defines the steps required to build and run software-intensive integrated product development (IPD) teams [1]. First, the engineers are trained precisely how to do quality work, use a defined process, and make and use process measurements. For engineers to use these methods on the job, they must have hands-on training, explanation of the methods, and experience using them

Figure 1. *How PSP and TSP provide IPD capabilities.*

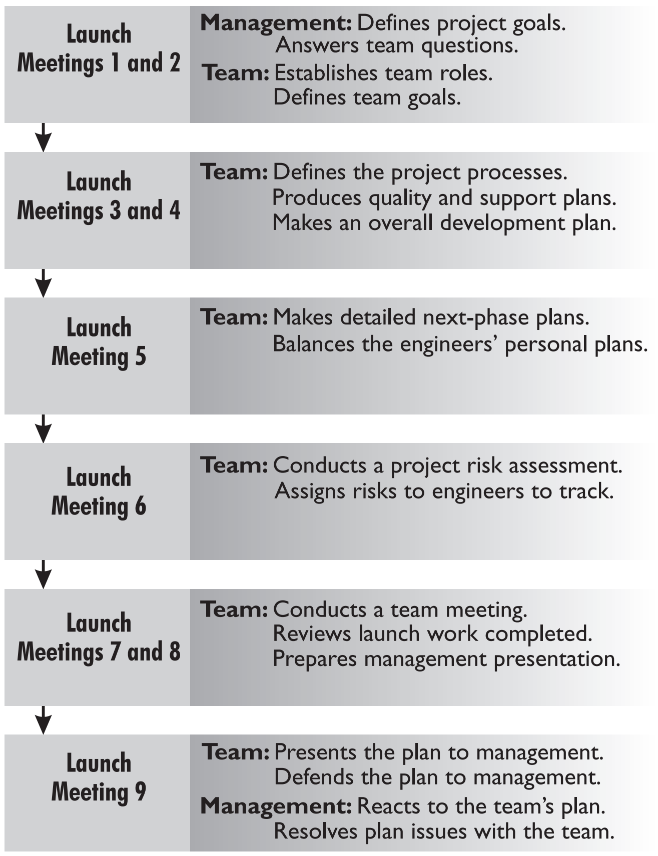


on realistic project-like exercises. This training is provided by an intensive 120-hour course that teaches the Personal Software Process (PSP)^{SM} [2,3,4,5]. Figure 1 shows how the PSP training and the TSP process provide the capabilities for integrated teamwork.

After acquiring basic process, planning, and quality management skills, engineers have the prerequisites to use the TSP. Every project then starts with a three-day TSP launch workshop, where engineers develop teamworking practices, establish goals, select roles, define processes, and make plans. A shorter two-day relaunch workshop is then repeated at the start of every major project phase. Because team members work directly on their project during the launch, these three days are part of the job and are not a training exercise.

Finally, the TSP provides the mechanisms to maintain an effective teamworking environment. This is done with structured weekly team meetings and periodic relaunch workshops. The team meeting is much like the football huddle: all members participate, and they focus on precisely what to do next. If the plan is working, they follow it. If it is not, they may decide to change it. The team meeting not only maintains effective team communication but also facilitates precise status tracking, provides a context for team decision making, and supports continuous risk tracking and project reporting. As in football, periodic "huddles" are important; if teams did not huddle, they would do a lot of running around but not win many games.

Figure 2. *The TSP launch process.*


Launch Meetings 1 and 2
Management: Defines project goals. Answers team questions.
Team: Establishes team roles. Defines team goals.
Launch Meetings 3 and 4
Team: Defines the project processes. Produces quality and support plans. Makes an overall development plan.
Launch Meeting 5
Team: Makes detailed next-phase plans. Balances the engineers' personal plans.
Launch Meeting 6
Team: Conducts a project risk assessment. Assigns risks to engineers to track.
Launch Meetings 7 and 8
Team: Conducts a team meeting. Reviews launch work completed. Prepares management presentation.
Launch Meeting 9
Team: Presents the plan to management. Defends the plan to management.
Management: Reacts to the team's plan. Resolves plan issues with the team.

The team relaunch is conducted at every principal project milestone. It serves to help the team evaluate and rebalance the project plan, reassess project risks, integrate new team members, reassign team roles, and re-emphasize the team's goals and charter. At the conclusion of each launch or relaunch, the team reviews its status, plans with management, and resolves any issues and problems.

What the TSP Provides

The TSP process provides a set of forms, scripts, and standards that lead the team through the process steps. Once they are PSP trained, engineers know how to develop and follow a defined process, and they understand how to use the process measures to consistently produce quality products. The PSP can be viewed as a language of process. Until engineers are reasonably fluent in this language, they generally are not able to follow the process and use its measures. PSP training provides the engineers the process fluency they need to use the TSP.

The TSP process also provides the guidance engineers need to work effectively in a team context. As shown in Figure 2, this is done during the three-day team launch. By following the launch process, the team members can quickly determine their own and everyone else's responsibilities, and they can readily track and coordinate their work with their teammates and other teams.

Because the TSP produces a large volume of data, managing and tracking the data can become a burden. The SEI has developed a support tool that helps engineers record and track TSP data. The initial tool support is in Microsoft Excel for Windows 95 and Windows NT. The TSP teams that have used this tool report that it substantially simplifies their data-gathering and reporting tasks. An enhanced tool is under development.

Engineering Support

During the launch and relaunch workshops, the team works as a unit to develop their process, quality, support, and project plans. These detailed plans identify and schedule the work for the next phase to the level of 10 task hours or fewer. Thus, the team members and their management know what tasks are to be done and when they are to be completed. In one example, Dave Webb, the TaskView team leader, needed to temporarily assign one engineer to help another project with a critical problem. By reviewing the detailed task schedule with the engineer, he precisely determined the impact of this reassignment and made workload adjustments to ensure that the project schedule was not affected.

The team as a unit also performs continuous risk management. In the launch and periodic relaunches, members do a complete project risk assessment. All risks are rated for likelihood and impact, and the more important risks are assigned to individual members for tracking. The assigned team members then develop mitigation plans for the immediate priority risks and monitor and report risk status in the weekly team meetings.

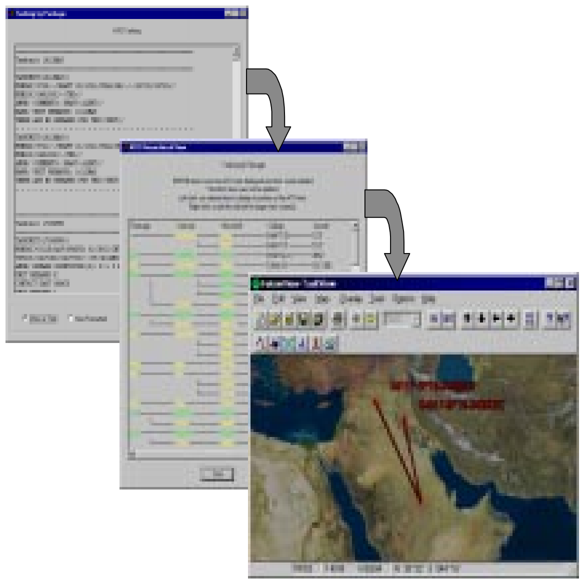


Figure 3. *TaskView converts complex ASCII text to tree structures to map routes.*

The TSP process helps working groups develop into cohesive and effective engineering teams. With defined and agreed-to goals and a process and plan to meet these goals, team members are more likely to submerge their personal problems and strive for the common objective. Efficiency is enhanced by the defined process, and communication is maintained by the weekly meetings of all team members. These meetings take less than one hour for teams of about 10 members. Team members review their role activities, planned vs. actual tasks completed, and risk status. Each member reports personal earned-value status, any needed team or management actions, and personal plans for the next period. These weekly meetings permit the team as a whole to periodically rebalance the workload, resolve issues, and make decisions.

TSP Status

The TSP process is being developed by the SEI, and it is currently under test by approximately 10 engineering groups and several dozen teams. Based on the experience to date, four TSP versions have been produced. The TSP has been used with teams as small as two engineers and with groups as large as 17. Some teams have been composed of software professionals, and others have also had hardware, systems, test, or other engineering participants. The project categories include maintenance, new product development, and product enhancement. System types have ranged from components of large commercial data-processing systems to embedded real-time controllers. TSP projects have covered proprietary product development, industrial software contracts, and military development and enhancement work.

Hill Air Force Base

The TaskView project was conducted by the Ogden Air Logistics Center, Technology and Industrial Support Directorate (TI), Software Engineering Division (TIS) at Hill Air Force Base, Utah. The TIS vision statement declares that they will provide "exceptional weapon system software and related hardware solutions and technology adoption expertise to enhance our nation's defense."

TIS is a high-maturity organization with a strong history of software process improvement. In March 1995, TIS was assessed as a CMM Level 3 organization, and the assessment conducted in July 1998 rated them at CMM Level 5. This is the first software organization in the Department of Defense (DoD) to receive this rating, and it is one of the few Level 5 software groups in the world.

The software products produced by TIS include operational flight programs for the F-16 Fighting Falcon aircraft, test program sets for F-16 automated test equipment, mission-planning software for a variety of aircraft, and avionics test-station software. TIS is also the home of the Software Technology Support Center (STSC), which provides technology adoption expertise to the DoD, sponsors the annual Software Technology Conference, and publishes CROSSTALK.

During the summer of 1996, TIS introduced the PSP to a small group of software engineers. Although the training was generally well received, use of the PSP in TIS started to decline as soon as the classes were completed. Soon, none of the engineers who had been instructed in PSP techniques was using them on the job. When asked why, the reason was almost unanimous: "PSP is extremely rigorous, and if no one is asking for my data, it's easier to do it the old way."

Although the TIS Software Engineering Process Group (SEPG) believes that PSP training accelerated CMM improvement work, members were concerned that the PSP methods were not being used. They therefore asked the SEI how to get engineers to consistently use PSP practices on the job. Because the TSP was then being designed to address this exact problem, the SEI suggested that TIS become involved in TSP pilot testing. TIS decided to do so, and this project is the result.

The TaskView Project

TIS chose the TaskView project as the TSP pilot. TaskView is a UNIX-based tool that parses an Air Tasking Order (ATO), which is a set of battle instructions for all aircraft involved in a strike, including fighters, bombers, and refuelers. As shown in Figure 3, it describes the flight plans, aircraft armament, and specific mission roles and tasks. Once the battle has been planned, a complex set of computer programs generates an ASCII text file that contains the ATO information. This ATO is then delivered electronically to each of the units participating in the strike.

Currently, the ATO is "broken out" manually— interpreted, sorted, and restructured—by the participating groups, who use hard copies and highlighters to mark their specific instructions. This is a laborious process that can take several hours. Once the information has been identified, the data must then be manually entered into mission-planning software tools for each unit, which provides ample opportunity for further mistakes. The TaskView tool parses the ATO and automatically "breaks out" (sorts and structures) the needed information in a few seconds. Additionally, TaskView can port data directly to mission-planning software tools, which greatly reduce the defects introduced during manual entry.

An initial prototype version of TaskView had been developed by another organization, and the TIS contract was to produce a product from this prototype, enhance it for a new ATO format, and port it from the UNIX environment to a PC Windows NT operating system.

TIS chose the TaskView project as a pilot for the TSP for several reasons:

- The team members were already PSP trained.
- TaskView was a small (under 20,000 lines of code [LOC]), short-duration (eight months) project from which results would be immediately apparent.
- The project manager for TaskView (Dave Webb) was an SEI-certified PSP instructor.

The TaskView project started a month before the introduction of TSP. The team had already been through the planning process required by TIS, and a detailed plan already existed before the first TSP launch. Since the TSP is designed to build on and augment an organization's existing process, the TaskView project could use the TIS Standard Engineering Process and tracking tools. When organizations do not have a fully defined process, the TSP launch process guides the team in defining and developing the needed process elements.

Using the TSP Process

The first TSP launch for the TaskView project was held at the end of February 1998. During the launch, we reviewed TSP concepts with the team and guided them through the project planning and tracking steps. The team spent about two and one-half days in this launch workshop.

Team Goals and Roles

During the project launch, the team members determined and documented the project goals. Some were high level, such as "delight our customers" and "be an effective pilot project for TSP in the Air Force and the DoD." More specific goals included "provide clean beta versions of TaskView to [the customer]" and "meet or exceed our quality plan." One important goal was to meet the customer's recent request that the TaskView project be delivered one month earlier than the original Sept. 30, 1998 commitment date.

Next, team members chose their personal team roles from among the TSP basic set: Customer Interface Manager, Design Manager, Implementation Manager, Planning Manager, Process Manager, Quality Manager, Support Manager, and Test Manager. Because of the limited size of the team, some members received more than one job. These roles were assigned so that when risks or issues arose, there would be a point of contact already designated and prepared to handle them. As usual, the official team leader had already been designated by management.

Detailed Planning

With the goals and roles determined, the team refined its existing project plan. The previously developed TaskView plan contained about three dozen work breakdown structure elements and tasks. During the TSP launch, the engineers produced a detailed list of more than 180 tasks. Using standard productivity rates, the team next estimated the task hours and the size of each task's product, usually in LOC. They also estimated each engineer's available task hours for each week of the project.

Task hours are hours spent working *only* on the tasks in the task list. Time spent in meetings, on the telephone, using E-mail, or engaged in any other activity that is not defined in the plan is not counted toward TSP task hours. Although these activities are necessary and are definitely work hours, they are not tracked as part of the project earned value. Based on the experiences of other TSP projects, the TaskView team estimated that in an engineer's standard 40-hour workweek, 20 hours would be an aggressive goal for task-related work.

The TSP Earned-Value Tool

TSP tools were then used to turn this top-down plan into an earned-value chart with a projected completion date. On the first run, the team and management were delighted to find that the new completion date projected by the top-down plan matched perfectly with the customer requirement for a one-month schedule acceleration.

Next, the software engineers were each given a copy of the task list and asked to estimate their personal work, using their own line of code and effort data. Such data are a product of the PSP course, which every engineer should complete before starting a TSP project. The TSP tool was then used to combine these individual estimates into a bottom-up estimate, also with earned value and a projected completion date. This estimate did not match the schedule requirements or the top-down estimate completed only a few hours earlier because some engineers were tasked more heavily than others. Because project schedules often slip if only one engineer is overburdened, the TSP launch process includes a workload-balancing step.

After workload balancing, the bottom-up schedule matched the top-down estimate and the customer's need. At this point, all engineers had a personal task and earned-value plan for which they individually had provided the estimates.

Risk Assessment and Mitigation

At the next TSP launch meeting, the TaskView team identified the risks associated with the project. They listed these risks in a brainstorming session, prioritized risk likelihood and impact, and assigned responsibility for mitigation and tracking. For example, the risk that "there will be a day-for-day slip in schedule if we do not receive the necessary header files by 3 March" was given a high likelihood and impact and assigned to the official team leader.

Fourteen risks were identified in this initial launch, of which seven were assigned to the team leader, and the balance were handled by team mem-

Module Number	Estimated New and Changed LOC	Actual New and Changed LOC	Percent Error*
1	1,500	1,656	10.40%
2	1,500	1,350	-10.00%
3	500	418	-16.40%
4	3,000	4,525	50.83%
5	1,000	973	-2.70%
6	500	1,067	113.40%
7	500	0	-100.00%
8	1,100	3,377	207.00%
9	1,500	848	-43.47%
10	500	956	91.20%
11	1,500	1,494	-0.40%
12	9	4	-55.56%
13	500	653	30.60%
14	unused	unused	unused
15	500	965	93.00%
16	1,177	2,973	152.59%
17	819	1131	38.10%
18	3,000	4,386	46.20%
Total	19,105	26,776	40.15%

Table 1. *TaskView estimated vs. actual LOC. *Note that underestimates are positive, and overestimates are negative.*

bers. The team leader also agreed to share responsibility with the engineers to track and mitigate the other management-related risks.

Management Review

The final launch activity was a management review of the team's launch results. Normally, such meetings provide the forum to resolve serious scheduling or resource issues. For TaskView, however, the management review reaffirmed the existing project commitments.

Tracking the Work

After the two-and-one-half-day TSP launch, the team started on the job. Using the PSP, the engineers tracked, in minutes, the time they spent on each task and process phase, recorded the defects found at every phase, and measured the sizes of the products they produced. The data were stored in the engineers' data tracker and in the TSP tracking tools. Thereafter, the team met weekly to review earned-value status, goals, risks, issues, and action items.

Within the next few weeks, it was evident that the team had a problem. The engineers were not achieving the 20 task hours per week they had planned. Their earned-value data, however, showed them to be on or ahead of schedule. From the data, the team found that there were two offsetting factors: Tasks had generally been overestimated, and it was much harder to achieve 20 task hours per week than had been expected. Even though the schedule impact to date had been minimal, this new understanding helped the team make better plans, and it showed where to focus to improve performance.

The Team Relaunch

In May 1998, we guided the TaskView team in assessing their progress and conducting a relaunch. The relaunch was necessary because the project was moving into its second phase, and the engineers felt a new plan was needed. This new plan would reflect lessons learned from the prior phase, more realistically address task hours, and include new tasks.

Although relaunch workshops normally take two days, this team was able to accomplish it in only one day. During this period, they replanned the project, refined their size and time estimates, adjusted their schedule to reflect 15 weekly task hours per engineer, and reassessed risks. Based on the cost, schedule, risk, and quality data, the overall project was judged to be ahead of plan. Because tasks had been generally accomplished with less effort than originally planned, some functions were completed early, whereas one important function planned for Phase 1 had slipped to Phase 2.

Because of the project's progress, TaskView could either return some money to the customer or add new functionality. The customer interface manager worked with the customer and found that new functionality was more important than cost reduction. Management then agreed to add more tasks and more people to the project. These new functions caused a modest schedule delay, so the customer interface manager reviewed the new functionality and schedule with the customer for approval. Since the planned delivery was still months away, the customer decided to accept the small schedule change in order to get the added functions.

Project Results

To determine the benefits of the TSP, TIS compared the TaskView pilot with similar projects that followed the organization's standard process. The project manager and the software engineers were also asked how the TSP had helped or hindered their personal work. Because TIS projects already routinely meet schedules, commitment performance was not an important factor in the analysis.

Estimating Accuracy

Use of the TSP was found to substantially improve size and effort estimating accuracy. During the first launch, TaskView was estimated to be 14,065 LOC. By the second launch, with the new functions, the total estimated size grew to 19,105

Phase	TIS	TSP
Requirements inspection	X	X
High-level design inspection	X	X
Detailed design personal review		X
Detailed design inspection	X	X
Personal code review		X
Compile	X	X
Code inspection	X	X
Functional test	X	X
Candidate evaluation (CPT&E)	X	X
System test (ERT)	X	X
Operational test and evaluation (acceptance test)	X	X
Operational usage (external)	X	X

Table 2. *TIS and TSP defect-removal process steps.*

LOC. When the TaskView project was completed, the final new and changed LOC for the project was 26,776, an underestimate of 40 percent. When the 9,455 LOC of added function were subtracted, the team's original 14,065 LOC size estimate had an error of 23 percent.

Table 1 shows the size estimates the engineers made during the second TSP launch. Module 7 took no new and changed code because the engineer reused an existing routine. Although some individual estimates were reasonably close, there was considerable variation. By using a sound statistically based method and their personal historical data, however, the engineers were able to make balanced estimates. This meant that, on average, they were as likely to estimate high as low. Because the errors in the individual estimates tended to compensate, the overall estimate was much more accurate than were the individual estimates. Team members believed that their large personal estimating errors were largely due to the lack of historical data for this kind of project. Future project estimates will benefit from the data gathered during this project and should be more accurate.

The TaskView effort estimates were originally made before the introduction of the TSP. At the first launch, the effort was again estimated to determine if the costs were appropriate and if the load was properly balanced among the engineers. By the second launch, it was obvious that effort had been overestimated; the project was able to meet earned-value goals with fewer task hours than had originally been expected. After including the customer-requested new functionality, the final delivery date was only two days later than the accelerated schedule, and the cost error was negligible.

Productivity

The TIS software process database contains the average productivity in LOC per man-hour for this team's prior project, and the average productivity for every project that used the TIS organizational process. Although the exact numbers are proprietary, the TaskView project increased productivity to 16 percent above the TIS average. These particular engineers increased their productivity to 123 percent above their previous project, or more than two times. Data on the relative productivity in LOC per programmer-hour for TaskView, the team's prior project, and the average of all TIS projects are shown in Figure 4. The TIS average is shown as 100 and TaskView as 116.

Productivity figures are impacted by many factors. Because TaskView and the team's prior project involved different languages, application domains, and development environments, the productivity improvement cannot be considered a measure of the TSP. The results do, however, suggest that the TSP improves productivity.

Quality Improvement

As shown in Table 2, the standard TIS process includes inspections (peer reviews) of all work products. The TSP adds a set of personal design and code reviews. One important question was whether the time spent doing these personal reviews was worthwhile. The TIS process typically removes about 13 defects for every thousand lines of code (KLOC) during design and code inspections. The rest must be found in test or by the user. With TSP, the TaskView project increased the yield of early defect removal by more than 60 percent by removing 21 defects per KLOC in both the reviews and the inspections. The benefits of this early attention to quality are apparent from the results of the later test phases.

Assuming the engineering process has rigorous testing criteria, an indicator of product and process quality is the time spent running tests. Generally, the fewer defects there are to be found, the less time is spent in test and the higher is the resulting product quality. The TIS process has three test phases, all with rigorous criteria, that must be completed before the product is passed to an external agency for operational testing: functional test, candidate evaluation, and system test. These phases are then followed by the customer's operational test and evaluation and then by operational usage. Typical TIS projects require 22 percent of the project schedule (in days) to perform the final two TIS test phases. The TaskView project, using TSP, sharply reduced this percentage to 2.7 percent. This is a schedule savings of nearly 20 percent. Only one high-priority defect was found in these last two test phases.

Data from the completed TaskView project show that the defect density at the functional testing phase was close to that normally achieved by other TIS projects only after all engineering testing

Table 3. *TaskView testing time. *Acceptance test is continuing but no defects have been reported to date.*

	TaskView	Project 1	Project 2	Project 3
Program Size – LOC	26,776	67,291	7,955	86,543
CPT&E Test Days	4	22	10	33
ERT (System Test) Days	2	41	13	59
Total Test Days	6	63	23	92
Test Days/KLOC	0.22	0.94	2.89	1.06
System Test Defects/KLOC	0.52	2.21	4.78	2.66
Acceptance Test Defects/KLOC	0*	N/A	1.89	0.07

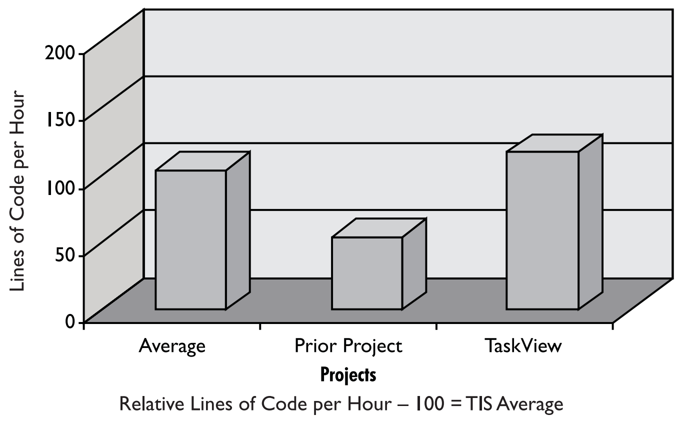


Figure 4. *Relative productivity.*

phases have been completed. In TaskView, to find only one high-priority defect in the TaskView product during system and operational testing is unprecedented for any TIS project.

Because of the improved quality from the TSP process, TaskView testing time was sharply reduced, as shown in Table 3. Here, the test data for the TaskView project are compared with three comparable prior projects. Although one could reduce testing time by running incomplete tests, the fact that the customer has so far reported no defects during acceptance test suggests that this was not the case. By using the TSP, TaskView not only produced a higher-quality product, it also took only one-eighth the testing time normally required for similar projects.

Qualitative Results

A critical question in introducing any new software engineering tool or technology is whether the engineers will use it. If the engineers do not like a tool or method, they will probably not use it, regardless of its effectiveness. To assess this issue, we privately asked all the TSP team members four questions:

- What do you believe are the advantages of the TSP?
- What do you believe are the disadvantages of the TSP?
- What about the TSP would you change?
- What about TSP would you keep the same?

Without knowing their teammates' responses, every team member said the TSP helped them form a closer, more effective team than any they had worked on before and that they would like to continue to use it. One team member said, "The TSP creates more group involvement. Everyone feels like they're more part of a group instead of a cog in a wheel. It forces team coordination to talk about and solve problems—there's no pigeonholing." Another team member said, "This really feels like a tight team. I was on the same team for a year [while working on another project] and didn't know the team members as well as I do now."

Another qualitative advantage expressed by multiple team members was increased effectiveness in project planning and tracking. "TSP gives you better insight into your current state," said one software engineer. "It provides better focus for the software developer on tasks to be done." Another TaskView team member summed up the planning and tracking benefits of TSP in this way: "Measuring progress helps generate progress."

The principal weakness the TaskView team mentioned was the need for better TSP tool support. Several members said that the tracking and earned-value support needed to be improved, and another suggested more automated data gathering and analysis. Work on TSP tool improvement has already begun at the SEI, and a newer, better version of the planning and tracking tool will soon be available.

The lead software engineer gave perhaps the best testimonial to the qualitative results of the TSP. When asked what he would not change about the TSP, he said, "I've seen a lot of benefits [from the TSP]. I'd like to see us continue to use it."

Conclusions

One of the fears many have about process improvement initiatives like the TSP is that the cost of doing extensive planning, personal reviews, and data gathering will increase the overall cost of the project. It is evident from the TaskView data, however, that the time spent performing these activities is more than made up by improved planning accuracy and reduced test time. As Philip Crosby once noted, "Quality is free." [6]

Perhaps the greatest change with the TSP is in the relationship between management and the engineers. To be most effective, engineers must be motivated and energetic; they need to be creative and concerned about the quality of their products, and they should enjoy their work and be personally committed to its success. This can only be achieved if management trusts the engineers to work effectively and the engineers trust their management to guide and support them.

Although trust is an essential element of effective teamwork, it must rest on more than mere faith. The engineers must follow appropriate methods and consistently strive for quality results. They must report on their progress and rapidly expose risks and problems. Similarly, management must recognize that the engineers generally know more about their detailed work than the managers, and they must rationally debate cost and schedule issues. Management also needs to ensure that the engineers consistently follow disciplined methods and that the teams do not develop interpersonal problems.

The TSP is designed to address these issues and show engineers and managers how to establish an environment in which effective teamwork is normal and natural. Because this will often require substantial attitude changes for the engineers and the managers, to introduce the TSP is a non-trivial step. As the TaskView data show, however, the TSP can produce extraordinary results. ◆

Acknowledgments

Being a leader and a coach for the TaskView team has been a rewarding experience for each of us. It would not, however, have been as rewarding or satisfying without a dedicated and hard-working team. For their support and cooperation, we thank Pattie Adkins, Keith Gregersen, Neil Hilton, Craig Jeske, Ken Raisor, Mark Riter, and Capt. David Tuma. We also enjoyed excellent support from Tresa Butler for configuration management, Pat Cosgriff for SEPG support, and Jim Van Buren of the STSC for PSP consultation.

For quality engineering work, consistent and informed management leadership is essential. For their trust in us and their willingness to support us in pioneering the early use of TSP in practice, we thank Dan Wynn, Robert Deru, Don Thomas, LaMar Nybo, and Eldon Jensen. Lt. Col. Jacob Thorn, the TaskView program manager at Eglin Air Force Base, Fla., also supported our process improvement initiatives. His dedication to quality and informed oversight made the job possible.

We also thank those who reviewed this article. Their comments and suggestions were a great help. Our particular thanks to Rushby Craig, Walter Donohoo, Linda Gates, John Goodenough, and Bill Peterson. Finally, the professional help and guidance of the CROSSTALK staff have, as always, been a great help.

About the Authors

David Webb has a bachelor's degree in electrical and computer engineering from Brigham Young University. He has worked for TIS for more than 11 years as a software engineer. Six of those years he spent as an F-16 Operational Flight Program software test engineer and system design engineer, three years as a member of the TIS SEPG, and two years as a technical program manager for TIS mission-planning software. He has participated in three CMM-Based Appraisals for Internal Process Improvement, including TIS's 1998 Level 5 assessment. He has also been certified by the SEI as a PSP course instructor.

OO-ALC/TISHD
6137 Wardleigh Road
Hill Air Force Base, UT 84056
Voice: 801-775-2916 DSN 775-2916
E-mail: webbda@software.hill.af.mil

Watts S. Humphrey is a fellow at the SEI at Carnegie Mellon University, which he joined in 1986. At the SEI, he established the Process Program, led initial development of the CMM, introduced the concepts of Software Process Assessment and Software Capability Evaluation, and most recently, the PSP and TSP. Prior to joining the SEI, he spent 27 years with IBM in various technical executive positions, including management of all IBM commercial software development and director of programming quality and process. He has a master's degree in physics from the Illinois Institute of Technology and in business administration from the University of Chicago. He is the 1993 recipient of the American Institute of Aeronautics and Astronautics Software Engineering Award and an honorary doctorate in software engineering from Embry Riddle Aeronautical University in 1998. His most recent books include *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), and *Introduction to the Personal Software Process* (1997).

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 412-268-6379
E-mail: watts@sei.cmu.edu

References

1. Humphrey, Watts S., "Three Dimensions of Process Improvement, Part III: The Team Process," *CROSSTALK*, Software Technology Support Center, Hill Air Force Base, Utah, April 1998, pp. 14-17.
2. Ferguson, Pat, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," *IEEE Computer*, May 1997, pp. 24-31.
3. Humphrey, Watts S., *A Discipline for Software Engineering*, Reading, Mass., Addison-Wesley, 1995.
4. Humphrey, Watts S., "Using a Defined and Measured Personal Software Process," *IEEE Software*, May 1996.
5. Humphrey, Watts S., "Three Dimensions of Process Improvement, Part II: The Personal Process," *CROSSTALK*, Software Technology Support Center, Hill Air Force Base, Utah, March 1998, pp. 13-15.
6. Crosby, Philip B., *Quality Is Free: The Art of Making Quality Certain*, McGraw-Hill, New York, 1979.

COMMUNITY, from page 2

as part of the employees' routine information flow. In addition to providing the infrastructure, organizations have to invest in hiring smart people and providing incentives for sharing information, then provide enough unstructured time to let people talk face to face. Such an environment will allow organizations to capitalize on their constellations of communities of practice.

People who appreciate the need for software knowledge management and who have the capacity to inspire or take the lead in providing the guidance and resources necessary to share information will continue to be invaluable. They can help any organization capitalize on opportunities by facilitating the enablers that are vital to our software community of practice. ◆

Notes

1. http://www.sei.cmu.edu
2. http://seir.sei.cmu.edu
3. http://www.stsc.hill.af.mil
4. http://esip.hill.af.mil
5. http://www.dacs.dtic.mil
6. http://www.spmn.com
7. http://www.deskbook.osd.mil
8. http://www.stc-online.org
9. http://www.sei.cmu.edu/products/events
10. http://www.sei.cmu.edu/cmm/cmms/cmms.integration.html

The Rosetta Stone
Making COCOMO 81 Estimates Work with COCOMO II

Donald J. Reifer, *Reifer Consultants, Inc.*
Barry W. Boehm and Sunita Chulani, *University of Southern California*

As part of our efforts to help Constructive Cost Model (COCOMO) users, we, the COCOMO research team at the Center for Software Engineering at the University of Southern California (USC), have developed the Rosetta Stone to convert COCOMO 81 files to run using the new COCOMO II software cost estimating model. The Rosetta Stone is extremely important because it allows users to update estimates made with the earlier version of the model so that they can take full advantage of the many new features incorporated into the COCOMO II package. This article describes both the Rosetta Stone and guidelines to make the job of conversion easy.

During the past few years, the COCOMO team at USC has been working to update the 1981 version of the COCOMO estimating model (COCOMO 81) [1]. The new version of the model, called COCOMO II, builds on the experiences that industrial affiliates of the USC Center for Software Engineering have had and addresses lifecycle processes and paradigms that have become popular since the original model was first introduced in 1981. These new paradigms include reuse-driven approaches, commercial-off-the-shelf lifecycle developments, component-based software engineering approaches, use of object-oriented methods, and other improvements to the way we do business stimulated by process improvement initiatives.

This article focuses attention on a tool we have developed to permit our users to update their original COCOMO 81 files so that they can be used with the COCOMO II model. We call the tool the Rosetta Stone because it is not unlike the black stone slab found in Egypt by French troops in 1799. On it were engraved three scripts (Greek, Demotic, and hieroglyphics), which enabled archaeologists to construct translations among the three languages. Our Rosetta Stone permits its users to translate files prepared with the original COCOMO 81 model to be compatible with COCOMO II.

Many of our affiliates thought creation of the Rosetta Stone was important because they had wanted to use the new version of the model to take advantage of its many advanced capabilities, including the COCOMO II package's autocalibration features. However, they could not make the move because they had files that required older versions of the model to run, e.g., COCOMO 81 and Ada-COCOMO. Others wanted to calibrate the new version of the model using their historical databases, but the new version of the model had a new structure, altered mathematics, and different parameters and parametric ratings. Under such circumstances, converting files was no easy task.

The COCOMO II Estimating Model

The major differences between COCOMO 81 and COCOMO II, why they are important, and cost driver definitions are summarized in Table 1. These changes are important because they reflect how the state of software engineering technology has matured during the past two decades. For example, programmers were submitting batch jobs when the COCOMO 81 model was first published. Turnaround time impacted their productivity. Therefore, a parameter TURN was used in the model to reflect the average wait programmers experienced before receiving their job back. Such a parameter is no longer important because most programmers have instant access to computational facilities through their workstations. Therefore, the parameter has been removed in the COCOMO II model.

The following summary highlights the major changes made to the original version of COCOMO 81 as COCOMO II was developed.

- COCOMO II addresses the following three phases of the spiral lifecycle: applications development, early design, and post-architecture.
- The three modes in the exponent are replaced by five scale factors.
- The following cost drivers were added to COCOMO II: DOCU, RUSE, PVOL, PEXP, LTEX, PCON, and SITE.
- The following cost drivers were deleted from the original COCOMO: VIRT, TURN, VEXP, LEXP, and MODP.
- The ratings for those cost drivers retained in COCOMO II were altered considerably to reflect more up-to-date calibrations.

The Rosetta Stone

As illustrated in Table 2, users need to convert factors in the COCOMO equations (such as the exponent, the size estimate, and the ratings for the cost drivers) from the original to the new version of the model. We suggest that users employ the following four steps to make the conversion so original files can be used with the COCOMO II model.

Update Size

The original COCOMO cost estimating model used deliverable source lines of

	COCOMO 81	COCOMO II
Model Structure	Single model that assumes you start with requirements allocated to software.	Three models that assume you progress through a spiral-type development to solidify your requirements, solidify the architecture, and reduce risk.
Mathematical Form of Effort Equation	Effort = $A(c_i)(Size)^{Exponent}$	Effort = $A(c_i)(Size)^{Exponent}$
Exponent	Exponent = Fixed constant selected as a function of mode. • Organic = 1.05 • Semidetached = 1.12 • Embedded = 1.20	Exponent = Variable established based on rating of five scale factors. • PREC: Precedentedness • FLEX: Development Flexibility • RESL: Architecture or Risk Resolution • TEAM: Team Cohesion • PMAT: Process Maturity
Size	SLOC (with extensions for FPs).	Object points, FPs, or SLOCs.
Cost Drivers (c_i)	15 drivers, each of which must be rated • RELY: Reliability • DATA: Database Size • CPLX: Complexity • TIME: Execution Time Constraint • STOR: Main Storage Constraint • VIRT: Virtual Machine Volatility • TURN: Turnaround Time • ACAP: Analyst Capability • PCAP: Programmer Capability • AEXP: Applications Experience • VEXP: Virtual Machine Experience • LEXP: Language Experience • TOOL: Use of Software Tools • MODP: Use of Modern Programming Techniques • SCED: Required Schedule	17 drivers, each of which must be rated • RELY: Reliability • DATA: Database Size • CPLX: Complexity • RUSE: Required Reusability • DOCU: Documentation • TIME: Execution Time Constraint • STOR: Main Storage Constraint • PVOL: Platform Volatility • ACAP: Analyst Capability • PCAP: Programmer Capability • AEXP: Applications Experience • PEXP: Platform Experience • LTEX: Language and Tool Experience • PCON: Personnel Continuity • TOOL: Use of Software Tools • SITE: Multisite Development • SCED: Required Schedule
Other Model Differences	Model based on • Linear reuse formula. • Assumption of reasonably stable requirements.	Has many other enhancements, including • Nonlinear reuse formula. • Reuse model that looks at effort needed to understand and assimilate. • Breakage ratings used to address requirements volatility. • Autocalibration features.

Table 1. *Model comparisons.*

code (DSI) as its measure of the size of the software job. DSI were originally represented by card images, e.g., includes all non-comment, nonblank carriage returns. COCOMO II uses the following three measures to bound the volume of work associated with a software job: source lines of code (SLOC), function points (FPs), and object points. SLOCs are counted using logical language statements per Software Engineering Institute (SEI) guidelines [2], e.g., `IF-THEN-ELSE, ELSE IF` is considered one, not two, statements.

Table 2 provides guidelines to convert size in DSI to SLOCs so that they can be used with the COCOMO II model. Whenever possible, we recommend using counts for the actual size of the file instead of the original estimate. Such practices allow you to correlate your actuals, e.g., the actual application size with the effort required to do the work associated with developing the software.

The reduction in size for COCOMO II estimates is attributable to COCOMO 81's need to convert card images to SLOC counts. As already noted, the pair `IF-THEN-ELSE` and `END IF` would be counted as two card images in COCOMO 81 and as a single source instruction in COCOMO II. The guidelines offered in Table 2 are based on statistical averages to simplify conversions; however, we encourage you to use your actuals if you have them.

The following are two common misconceptions about COCOMO's use of SLOC and FPs:

- *Misconception 1: COCOMO does not support the use of FPs* – FP versions of COCOMO have been available since the Before You Leap commercial COCOMO software package implementation in 1987. As noted in Table 1, COCOMO II supports use of either SLOC or FP metrics. In both cases, this is done via "backfiring" tables, which permit you to convert FPs to SLOCs at different levels.
- *Misconception 2: Although it is irresponsible to use SLOC as a general productivity metric, it is not irresponsible to use FP as a general sizing parameter for estimation* – This misconception breaks down into the two following cases.
  - Your organization uses different language levels to develop software. In this case, it is irresponsible to use SLOC as your productivity metric, since you get higher productivity per SLOC at higher language levels; however, it also is irresponsible to use FP as a general sizing metric because pure FP will generate the same cost (or schedule or quality) estimate for a program with the same functionality developed

COCOMO 81	COCOMO II
DSI • Second-Generation Languages • Third-Generation Languages • Fourth-Generation Languages • Object-oriented Languages	SLOC [3] • Reduce DSI by 35 percent. • Reduce DSI by 25 percent. • Reduce DSI by 40 percent. • Reduce DSI by 30 percent.
Function Points	Use the expansion factors developed by Capers Jones [4] to determine equivalent SLOCs.
Feature Points	Use the expansion factors developed by Capers Jones to determine equivalent SLOCs.

Table 2. *Converting size estimates.*

using different language levels. This is clearly wrong. To get responsible results in this case, FP-based estimation models need to use some form of backfiring to account for the difference in language level.

- Your organization always uses the same programming language (level). Here, it is responsible to use pure FP as your sizing metric for estimation. But it also is responsible to use SLOC as your productivity metric. Both metrics work in practice.

Convert Exponent

Convert the original COCOMO 81 modes to scale factor settings using the Rosetta Stone values in Table 3. Then, adjust the ratings to reflect the actual situation. For example, the Rosetta Stone rates process maturity (PMAT) low because most projects using COCOMO 81 are assumed to have been at Level 1 on the SEI process maturity scale [5]. However, the project's actual rating may have been higher and an adjustment may be in order.

An exception is the PMAT scale factor, which replaces the COCOMO 81 Modern Programming Practices (MODP) multiplicative cost driver. As seen in Table 4, MODP ratings of very low (VL) or low (L) translate into a PMAT rating of VL or a low level on the Software Capability Maturity Model scale. An MODP rating of normal (N) translates into a PMAT rating of L or a high Level 1. An MODP rating of high (H) or very high (VH) translates into a PMAT rating of N or CMM Level 2. As with the other factors, if you know that the project's actual rating was different from the one provided by the Rosetta Stone, use the actual value.

Table 3. *Model scale factor conversion ratings.*

Mode and Scale Factors	Organic	Semidetached	Embedded
Precedentedness (PREC)	XH	H	L
Development Flexibility (FLEX)	XH	H	L
Architecture and Risk Resolution (RESL)	XH	H	L
Team Cohesion (TEAM)	XH	VH	N
Process Maturity (PMAT)	L	L	L

The movement from modes to scale factors represents a major change in the model. To determine the economies and diseconomies of scale, five factors have been introduced. Because each of these factors can influence the power to which size is raised in the COCOMO equation, they can have a profound impact on cost and productivity. For example, to increase the rating from H to VH in these parameters can introduce as much as a 6 percent swing in the resulting resource estimate. Most of these factors are modern in their derivation. For example, the concept of process maturity was not in its formative stages when the original COCOMO 81 model was published. In addition, the final three factors, RESL, TEAM, and PMAT, show how an organization can exercise management control over its diseconomies of scale. Finally, the first two, PREC and FLEX, are the less controllable factors contributing to COCOMO 81 modes or interactions.

Rate Cost Drivers

The trickiest part of the conversion is the cost drivers. Cost drivers are parameters to which cost is sensitive. For example, as with the scale factors, you would expect that use of experienced staff would make a software development less expensive; otherwise, why use them? Because the new version of the model uses altered drivers, the Rosetta Stone conversion guidelines outlined in Table 4 are important. For those interested in more details about the cost drivers, we suggest you refer to the COCOMO II Model Definition Manual [6]. Again, the ratings need to be adjusted to reflect what actually happened on the project. For example, the original estimate may have assumed that analyst capability was very high; however, the caliber of analysts actually assigned might have been nominal because key employees were not available to the project when they were needed.

Users should take advantage of their knowledge of what occurred on the project to make their estimates more reflective of what really went on as the application was developed. Use of such knowledge can improve the credibility and accuracy of their estimates.

The TURN and TOOL rating scales have been affected by technology changes since 1981. Today, virtually everyone uses interactive workstations to develop software. TURN has therefore been dropped from COCOMO II and its calibration assumes the TURN rating is L. Table 5 provides alternative multipliers for other COCOMO 81 TURN ratings.

The tool suites available in the 1990s far exceed the COCOMO 81 VH TOOL rating, and virtually no projects operate at the COCOMO 81 VL or L TOOL levels. COCOMO II has shifted the TOOL rating scale two levels higher so that a COCOMO 81 N TOOL rating corresponds to a VL COCOMO II TOOL rating. Figure 5 also provides a

COCOMO 81 Drivers	COCOMO II Drivers	Conversion Factors
RELY	RELY	None. Rate the same or the actual.
DATA	DATA	None. Rate the same or the actual.
CPLX	CPLX	None. Rate the same or the actual.
TIME	TIME	None. Rate the same or the actual.
STOR	STOR	None. Rate the same or the actual.
VIRT	PVOL	None. Rate the same or the actual.
TURN		Use values in Table 5.
ACAP	ACAP	None. Rate the same or the actual.
PCAP	PCAP	None. Rate the same or the actual.
VEXP	PEXP	None. Rate the same or the actual.
AEXP	AEXP	None. Rate the same or the actual.
LEXP	LTEX	None. Rate the same or the actual.
TOOL	TOOL	Use values in Table 5.
MODP	Adjust PMAT settings.	If MODP is rated • VL or L, set PMAT to VL. • N, set PMAT to L. • H or VH, set PMAT to N.
SCED	SCED	None. Rate the same or the actual.
	RUSE	Set to N or actual, if available.
	DOCU	If Mode: = Organic, set to L. = Semidetached, set to N. = Embedded, set to H.
	PCON	Set to N or actual, if available.
	SITE	Set to H or actual, if available.

Table 4. *Cost drivers conversions.*

set of COCOMO II multipliers corresponding to COCOMO 81 project ratings.

Some implementations of COCOMO II, such as the USC COCOMO II package, provide slots for extra user-defined cost drivers. The values in Figure 5 can be put into those slots (if you do this, use an N rating in the normal COCOMO II TOOL slot).

To learn more about the cost drivers and their ratings, refer to the USC Web site (http://sunset.usc.edu/COCOMOII) or several of the Center for Software Engineering's other publications [7, 8]. Because the goal of this article is to present the Rosetta Stone, we did not think it was necessary to go into the details of the model and an explanation of its many parameters.

Experimental Accuracy

To assess the accuracy of the translations, the team used the Rosetta Stone to convert 89 projects. These projects were clustered subsets of the databases we used for model calibration. Clusters were domain-specific. We updated our estimates using actuals whenever we could. We then used the autocalibration feature of the USC COCOMO II package to develop a constant for the effort equation, e.g., the A in the equation $\text{Effort} = A(\text{size})^P$. Finally, we compared our estimates to actuals and computed the relative error as a function of the following cases.

- Using the Rosetta Stone with no adjustments.
- Using the Rosetta Stone with knowledge-base adjustments, e.g., updating the estimate files with actuals when available.
- Using the Rosetta Stone with knowledge-base adjustments and domain clustering, e.g., segmenting the data based on organization or application area.

The results of these analyses, which are summarized in Table 6, were extremely positive. They show that we can achieve an acceptable degree of estimating accuracy when using the Rosetta Stone to convert COCOMO 81 files to run with the COCOMO II software cost model.

Summary and Conclusions

The Rosetta Stone was developed to provide its users with a process and a tool to convert their original COCOMO 81 files so that they can be used with the new COCOMO II estimating model. The Stone represents a starting point for such efforts. It does not replace the need to understand either the scope of the estimate or the changes that occurred as the

Table 5. *TURN and TOOL adjustments.*

COCOMO 81 Rating	VL	L	N	H	VH
COCOMO II Multiplier: TURN		1.00	1.15	1.23	1.32
COCOMO II Multiplier: TOOL			1.24	1.10	1.00

Table 6. *Estimate accuracy analysis results.*

Cases	Accuracy (Relative Error)
Using the COCOMO II model as calibrated.	Estimates are within 25 percent of actuals 68 percent of the time.
Using the COCOMO II model as calibrated using developer or domain clustering.	Estimates are within 25 percent of actuals 76 percent of the time.
Using the Rosetta Stone with no adjustments.	Estimates are within 25 percent of actuals 60 percent of the time.
Using the Rosetta Stone with knowledge-base adjustments.	Estimates are within 25 percent of actuals 68 percent of the time.
Using the Rosetta Stone with knowledge-base adjustments and domain clustering.	Estimates are within 25 percent of actuals 74 percent of the time.

project unfolded. Rather, the Stone takes these considerations into account as you update its knowledge base with actuals.

The value of the Rosetta Stone was demonstrated convincingly based on an accuracy analysis of an 89-project database. As expected, the accuracy increased as we adjusted the estimates using actuals and looked at results based on domain segmentations. We are encouraged by the results. We plan to continue our efforts to provide structure and support for such conversion efforts. ◆

References

1. Boehm, B., *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
2. Park, R., "Software Size Measurement: A Framework for Counting Source Statements," CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, Pa., 1992.
3. Reifer, D., personal correspondence, 1998.
4. Jones, C., *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, New York, 1992.
5. Paulk, M., C. Weber, B. Curtis, and M. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, Mass., 1995.
6. Boehm, B., et al., *COCOMO II Model Definition Manual, Version 1.4*, University of Southern California, Los Angeles, Calif., 1997.
7. Boehm, B., et al., "The COCOMO 2.0 Software Cost Estimation Model," *American Programmer*, July 1996, pp. 2-17.
8. Clark, B. and D. Reifer, "The Rosetta Stone: Making Your COCOMO Estimates Work with COCOMO II," *Software Technology Conference*, Salt Lake City, Utah, 1998.

About the Authors

Donald J. Reifer is a leading figure in software engineering and management, with over 30 years progressive experience in government and industry. He has been chief of the Ada Joint Program Office, technical adviser to the Center for Software, and director of the Department of Defense (DoD) Software Reuse Initiative under an Intergovernmental Personnel Act assignment with the Defense Information Systems Agency. He is currently president of RCI, a small consulting firm servicing Fortune 500 companies, and he is a visiting associate at USC, where he serves on the COCOMO team. He has a bachelor's degree in electrical engineering, a master's degree in operations research, and a certificate in business management (master's equivalent). His many honors include the Secretary of Defense's medal for Outstanding Public Service, the NASA Distinguished Service Medal, the Freiman Award, and the Hughes Aircraft Fellowship. He has over 100 publications, including his popular *IEEE Software Management Tutorial* and a new Wiley book entitled *Practical Software Reuse*.

Reifer Consultants, Inc.
P.O. Box 4046
Torrance, CA 90505
Voice: 310-530-4493
E-mail: d.reifer@ieee.org

Barry W. Boehm is considered one of the fathers of the field of software engineering. He is currently director of the Center for Software Engineering at USC, and for many years directed key technology offices in the DoD, TRW, and Rand Corporation. His contributions to software engineering include COCOMO, the spiral model of the software process, the Theory W approach to software management and requirements determination, and the TRW Software Productivity System and Quantum Leap advanced software engineering environments. His current software research interests include process modeling, requirements engineering, architectures, metrics and cost models, engineering environments, and knowledge-based engineering.

Boehm has a bachelor's degree from Harvard University and a master's degree and a doctorate from the University of California at Los Angeles, all in mathematics. He has served on the board of several scientific journals and has served as chairman of numerous prominent engineering society committees. He is the recipient of many of software engineering's highest awards. He is an American Institute of Aeronautics and Astronautics Fellow, an Association for Computing Machinery Fellow, an Institute of Electrical and Electronics Engineers Fellow, and a member of the National Academy of Engineering.

Center for Software Engineering
University of Southern California
941 West 37th Place
Los Angeles, CA 90089
Voice: 213-740-8163
E-mail: boehm@sunset.usc.edu

Sunita Chulani is a research assistant at the Center for Software Engineering at the University of Southern California. She is an active participant on the COCOMO II research team and is working on a Bayesian approach to data analysis and model calibration. She is also working on a cost and quality model that will be an extension to the existing COCOMO II model. Her main interests include software process improvement with statistical process control, software reliability modeling, risk assessment, software cost estimation, and software metrics. She has a bachelor's degree in computer engineering from Bombay University and a master's degree in computer science from USC. She is currently a doctoral candidate at the Center for Software Engineering at USC.

Center for Software Engineering
University of Southern California
941 West 37th Place
Los Angeles, CA 90089
Voice: 213-740-6470
E-mail: sdevnani@sunset.usc.edu

Writing Effective Natural Language Requirements Specifications

William M. Wilson

This article details writing practices that will produce a stronger requirements specification document by avoiding three common documentation problems. Examples of these problems and the recommended solutions presented in this article were derived by analyzing 40 approved NASA requirements specification documents.

The system or software requirements specification (SRS) document is the first definitive representation of the capability that the provider is to deliver to the user or acquirer. The SRS document becomes the basis for all a project's subsequent management, engineering, and assurance activities. As such, it is a strong source of potential risks that could adversely impact the project's resources, schedules, and products. Because of the criticality of the SRS, it is important to prevent or correct shortcomings in both the form and content of the SRS document before it is established as a project baseline.

A study conducted by the Software Assurance Technology Center (SATC) located at NASA's Goddard Space Flight Center (GSFC) found that 40 approved SRS documents contained many instances of three common weaknesses. Deficiencies were found in

- the organization of requirement information.
- the structure of individual requirement statements.
- the language used to express requirements.

These defects can be prevented through a more disciplined and consistent approach to document design, formulation of specification statements, and selection of key words and phrases.

Organizing Requirements

To develop, deliver, and install system or software capability that successfully satisfies the expectations and needs of the user, the provider of the capability must have access to a wide variety of information. In addition to the requirements that prescribe a solution to the user's needs, descriptions of the user's current and future operational environments and a definition of the transition from one environment to the other must be provided. In some instances, support considerations make it necessary to dictate restrictions on the development environment and limit the technology content of the delivered capability.

Detail and Consistency

The problem of organizing the requirements information is compounded by the need for specific topics to be addressed in detail and in a manner that enhances comprehension and minimizes redundancy. The Institute of Electrical and Electronics Engineers (IEEE) [1] recommends that an SRS address each of the following topics.

- Interfaces.
- Functional capabilities.
- Performance levels.
- Data structures and elements.
- Safety.
- Reliability.
- Security and privacy.
- Quality.
- Constraints and limitations.

The first four topics address engineering requirements associated with individual elements of the needed capability. The last five topics address quality requirements that cross all aspects of the needed capability. These topics are not isolated subjects; they are coupled in various ways. Functions, interfaces, and data are closely linked. The question may arise: Should the section of the SRS that prescribes the requirements for a particular function include interfaces and data specifications, or should it point to sections of the SRS devoted to these topics? The first alternative distributes similar information across several sections of the document and creates problems in maintaining the document. The second alternative breaks the reader's train of thought when the referenced information is needed. The basic structural issue is how to organize these topics so that relationships and necessary detail can be stated clearly and succinctly.

Data Item Descriptions (DIDs) refine the SRS's design in most documentation schemes and are used to establish generic design solutions for each type of document. As with most general solutions, DIDs only resolve issues at the highest level of organization. Documentation standards developed by both the Department of Defense (DoD) [2] and NASA [3] include several DIDs to specify systems at various levels and from different aspects. The scope and number of DIDs included in these standards are intended to facilitate the documentation of large programs and projects. For smaller projects, these schemes are burdensome. Smaller projects tend to use fewer documents to decrease costs and facilitate information control.

Both DoD and NASA documentation standards provide for adaptation of DIDs to meet the needs of a particular project: DoD provides specific guidance for tailoring its DIDs, NASA provides guidance for tailoring and "rolling up" the concept, requirements, design, and other documents into a single volume.

Tailoring DIDs is a design activity. Its potential impact on a project's success is significant and should be undertaken with the same importance given to the design of any engineering product. The final design of the SRS must be a structure of sections and subparagraphs that encompass and address the concerns of all project participants and organizational stakeholders. The SRS structure must facilitate everyone's understanding of the totality of required capability, the particular attributes of a capability, and how their areas of responsibility will be affected by the capability. Sections of the documents must be constructed in light of the cohesiveness, coupling, complexity, and consistency necessary to achieve a balance between comprehensibility and completeness.

Information should never be arbitrarily grouped together—it makes the document difficult to understand and to maintain. Descriptions of the conditions and situations that the required capability will encounter must be located with the prescription of its required response; however, the description and prescription must be kept distinct from one another. Requirements that are parts of a single functional capability must be grouped together, e.g., functions that are connected in series by output-to-input relationships should appear in the SRS in the same sequence, if possible. Functions that share common inputs and outputs should be addressed within the same section of the SRS. If several processes must be accomplished in the same time frame, their specifications must be tied together by the document's structure to make this commonality clear. Similar functions need to be distinguished from one another but the similarities also need to be emphasized. Most of these restrictions can be satisfied by combining a requirements identification scheme that consistently uses similar numbers to number similar things and by using a writing style that uses short declarative sentences.

Both the DoD SRS DID, DPSC-8-1433, [2] and NASA-DID-P200, [4] provide an excellent starting point for the organization of a requirements document. Most structures provide for most environmental subjects as well as the nine essential topics of requirements information. It is highly desirable to have the same topics identified with the same number in related documents such as the Operational Concept Description, System/Subsystem Specification, SRS, and Software Product Specification; therefore, care must be taken to ensure that the integrity of paragraph numbering is maintained when the structure of the DID is shortened or extended. If not, any correlation of like information across the set of documents will be lost. Both NASA and DoD documentation tailoring instructions address this problem; the crux of their tailoring instructions is to not change subject numbers.

Unnecessary sections should be "stubbed" with an "N/A" as close to the main section of the document as possible. Stubbing retains the topic's title and identification number. Cutting sections removes the topic and reassigns its identification number to the following topic. This would apply through the rest of the document's numbers and destroy the document's congruence with the rest of the document set. New topics should extend the numbering established by the DID by adding a node (section) to the document tree that is appropriate to the new subject. They should be inserted as the last branch or leaf (at the end) and given the node's next available sequential number.

Statement Structuring

Poorly structured specification statements result in confusing requirements that are prone to incorrect interpretations. If a specification statement contains three or more punctuation marks, it probably needs to be restructured.

An example of a specification that is a prime candidate for restructuring follows:

> 3.1 The XYZ system shall provide variance/comparative information that is timely, itemized in sufficient detail so that important individual variances are not obscured by other variances, pinpoints the source of each variance, and indicates the area of investigation that will maximize overall benefits.

It is much easier to read when structured as follows:

3.1 The XYZ system shall provide variance/comparative information.

  3.1.1 Variance/comparative information shall be timely.

  3.1.2 Variance/comparative information shall be itemized in sufficient detail to do the following:

    3.1.2.1 Prevent important individual variances from being obscured by other variances.

    3.1.2.2 Pinpoint the source of each variance.

    3.1.2.3 Indicate the area of investigation that will maximize overall benefits.

Each specification statement consists of four basic structural elements—entities, actions, events, and conditions. These elements can be used or modified by various cases such as the following:

- Owner.
- Actor.
- Target.
- Constraint.
- Owned.
- Action.
- Object.
- Localization.

The recommended model for a specification statement's structure is as follows:

- Localization.
- Actor/Owner.
- Action.
- Target/Owner.
- Constraint.

For example, "When three or more star trackers lose reference stars, the spacecraft shall immediately align its main axis on the earth-sun line unless the optical instrument's cover is not closed."

- Localization – *When three or more star trackers lose reference stars.*
- Actor/Owner – *Spacecraft.*
- Action – *Align.*
- Target/Owned – *Main axis.*
- Constraint – *Unless the optical instrument's cover is not closed.*

Problems with Natural Language

Natural language's extensive vocabulary and commonly understood syntax facilitate communication and make it an inviting choice to express requirements. The informality of the language also makes it relatively easy to specify high-level general requirements when precise details are not yet known. However, because of differences among formal, colloquial, and popular definitions of words and phrases and the effort required to produce detailed information, these same attributes also contribute to documentation problems. The use of natural language to prescribe complex, dynamic systems has at least three common and severe problems: ambiguity, inaccuracy, and inconsistency [5].

The precise meaning of many words and phrases depends entirely on the context in which they are used. For example, Webster's New World Dictionary identifies three variations in meaning for the word "align," 17 for "measure," and four for "delete." Even though the words "error," "fault," and "failure" have been precisely defined by the IEEE [5], they frequently are used incorrectly in specifications.

Attention must be given to the role of each word and phrase when formulating specification statements. Words and phrases that are carelessly selected or carelessly placed produce statements that are ambiguous and imprecise.

The most simple word that is appropriate to its intended purpose in the specification is the one to use. The word "hide" is defined as "to put out of sight." The word "obscure" is defined as "lacking light or dim." Do not use *obscure* if you mean *hide*—the rules for the game "obscure and seek" are not well known.

Use the correct imperative and use it consistently. Remember that the word "shall" prescribes, "will" describes, "must" and "must not" constrain, and "should" suggests. Avoid weak phrases such as "as a minimum," "be able to," "capable of," and "not limited to." These phrases are subject to different interpretations and also set the stage for future changes to the requirements.

Do not use words or terms that give the provider an option regarding the extent to which the requirement is to be satisfied, such as "may," "if required," "as appropriate," or "if practical." Do not use generalities when numbers are required, for example, "large," "rapid," "many," "timely," "most," and "close." Avoid imprecise words that have relative meanings such as "easy," "normal," "adequate," or "effective."

The use of imprecise terms usually indicates that the specifications author was either lazy, incompetent, or did not have sufficient time to determine the exact requirements. Some writers seem to be afraid that their audience will be bored or will think them lazy if they use simple words and repeat themselves. When writing documents or software, being too fancy complicates things and make the resulting products hard to understand.

The previously given example specification could be further strengthened through a better selection of words and phrases.

3.1 The XYZ system shall provide variance/comparative information.
  3.1.1 Variance/comparative information shall be provided at the end of every reporting cycle.
  3.1.2 Variance/comparative information shall include the data necessary to
    3.1.2.1 Prevent important individual variances from being hidden by other variances.
    3.1.2.2 Pinpoint the source of each variance.
    3.1.2.3 Determine the frequency and severity of each variance.

Serious problems will always result from specifications statements such as "The system shall be user friendly and provide adequate resources to meet the user's operational needs." What is considered to be "user friendly," "adequate resources," and "user's operational needs" must be defined in detail through specification or by reference to an existing system that has the required characteristics.

Natural language often entices authors to write "stream of consciousness" specification statements that are difficult to understand, for example, "Users attempting to access the ABC database should be reminded by a system message that will be acknowledged and on page headings on all reports that the data is sensitive, and access is limited by their system privileges."

Restructured as shown below, this requirement, although longer, is easier to comprehend.

4.4 The system shall notify users attempting to access the ABC database that
  a. The ABC data is classified as "sensitive."
  b. Access to the ABC data is limited to that allowed by the user's system privileges.
  c. Page headings on all reports generated using the ABC database must state that the report contains "sensitive" information.
  4.4.1 The system shall require the user to acknowledge the notification before being allowed to access the ABC database.

Summary and Conclusion

When natural language is used to specify requirements, several things must be kept in mind.

- The SRS is the medium to express requirements that have been identified and defined. The SRS's DID is not an outline for a method to derive requirements.
- The SRS is a software item and as such should be a product that is engineered to satisfy the project's needs.
- Begin the design process with the appropriate SRS DID.
- Use simple sentence structures and select words and phrases based on their formal definitions, not on how popular culture defines them.
- The SRS must be understandable. It does not have to be interesting. Aspire to be a good engineer, not a literary artist. ◆

About the Author

William M. Wilson is a retired principal systems engineering consultant formerly with the Software Assurance Technology Center (SATC). He has 40 years of professional engineering experience with NASA, DoD, and industry. He is a recognized author and instructor of software safety and reliability courses, an authority on the specification of software requirements, and a trained lead auditor under the TickIT software certification scheme. Before

The SSG Systems Engineering Process

This brief overview of the Standard Systems Group (SSG) Systems Engineering Process (SEP) summarizes the primary objectives of the SEP. For complete information, see http://web1.ssg.gunter.af.mil/sep/SEPver40/ssddview.html.

In May 1997, the SSG at Maxwell Air Force Base, Gunter Annex in Montgomery, Ala. was rated Level 3 according to the Software Engineering Institute Capability Maturity Model. SSG is one of the larger, more diverse government agencies to achieve this distinction. Continuous, sustained process improvements led to this maturity level, and the method by which it was achieved is embodied in the SEP, now in Version 4. A combination of management and engineering activities composes this standard organizational process that can be tailored to address project specifics.

The SEP is a pragmatic, disciplined approach to software systems engineering. It describes the essential elements of an organization's systems engineering process that must exist to ensure good systems engineering.

The SEP's goals for a product are to

- Meet customer's functional objectives.
- Minimize defects.
- Enhance look and feel of having been built by one person, though it does not depend on one person for maintenance.
- Reduce risk; eliminate rework.
- Improve predictable schedule and cost.
- Provide development insight.
- Enhance maintainability.
- Introduce industry best practices.
- Operationalize policies and directives.

Success in a market-driven and contractually negotiated market is often determined by how efficiently an organization translates customer needs into a product that meets those needs. Good systems engineering is key to that activity, and the SEP provides a way to define, measure, repeat, and enhance performance. The SEP acts as a framework to which continuous process improvement can be added.

Under the SEP, projects and systems experience productivity improvements of 200 percent to 300 percent, a hundredfold reduction in post-release defects, less overtime and fewer crises, a return on investment of up to a ratio of 7-to-1, reduced long-term sustainment costs, and improved interoperability. The employees also are able to feel more competitive.

The greatest benefit of the SEP is that it increases the ability to meet customer cost, schedule, and performance expectations.

Point of Contact: Barry Morton
SSG Software Engineering Process Group Facilitator
Voice: 334-416-3547 DSN 596-3547
E-mail: MortonB@ssg.gunter.af.mil

joining the SATC, he was vice president of Quong and Associates, a consulting firm specializing in quality engineering and assurance practices. He was responsible for aerospace industry software quality assurance standards and procedures. While manager of software engineering assurance in the Office of the Chief Engineer at NASA Headquarters, he established and directed the Software Management and Assurance Program, which produced NASA's first agency-wide software policies and standards. As a member of the Defense Communications Agency's National Military Command Systems Engineering Directorate, he was the project manager and systems engineer for the acquisition and development of several first-generation strategic command, control, and communications systems that support the National Command Authority. He has a bachelor's degree in electrical engineering. He is a member of the IEEE Computer Society, the Association for Computing Machinery, and the American Society for Quality Control.

Point of Contact: Linda H. Rosenberg
Goddard Space Flight Center
Code 300.1, Building 6
Greenbelt, MD 20771
E-mail: Linda.H.Rosenberg.1@gsfc.nasa.gov

References

1. IEEE Standard 830-1993, Recommended Practice for Software Requirements Specifications, Dec. 2, 1995.
2. MIL-STD-498, Software Development and Documentation, Dec. 5, 1994 (http://scpo.nosc.n&498.html).
3. Ganska, Ralph, John Grotzky, Jack Rubinstein, and Jim Van Buren, *Requirements Engineering and Design Technology Report*, Software Technology Support Center, Hill Air Force Base, Utah, October 1995.
4. NASA-STD-2100-91, *NASA Software Documentation Standard*, NASA Headquarters Software Engineering Program, July 29, 1991 (http://satc.gsfc.nasa.gov/assure/docstd.html).
5. Wilson, William, Linda H. Rosenberg, and Lawrence E. Hyatt, "Automated Quality Analysis of Natural Language Requirement Specifications," *Pacific Northwest Software Quality Conference Proceedings*, October 1996, pp. 140-151 (http://sate.gdcnasa.gov/SATC/PAPERS/PNQ/pncl.htnil).
6. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

Software Product Lines
A New Paradigm for the New Century

Paul Clements
Software Engineering Institute

Software developed as a product line promises to be a dominant development paradigm for the new century, one that the Department of Defense (DoD) can leverage when acquiring software-intensive systems. This article discusses the advantages of product lines, uncovers some of their pitfalls, and shows by example the kinds of successes the organizations can enjoy.

Imagine turning out a 1.5 million-line Ada command and control system for a Navy frigate warship. The system is hard real-time, fault-tolerant, and highly distributed, running on 70 separate processors on 30 different local area network nodes scattered all over the ship. It must interface with radars and other sensors, missile and torpedo launchers, and other complicated devices. The human-computer interface is complex and highly demanding. In this application, quality is everything: The system must be robust, reliable, and avoid a host of performance, distribution, communication, and other errors.

Now suppose that you have not one of these systems to build but several. Your marketing department has succeeded beyond your wildest dreams. Navies from all over the world have ordered your command and control system. Now, your software must run on almost a dozen different ship classes including a submarine, systems that are drastically separate: The end users speak different languages (therefore, the human-computer interface requirements are extremely different), the ships are laid out differently, have different numbers of processors and nodes, and different fault tolerance requirements, different weapons systems and sensors, and different computers and operating systems. But quality remains crucial in all of them.

Suppose you are the manager for this megaproject. Do you panic? Do you resign? Run to a third-world country? What if you could produce each one of the systems for a fraction of the cost and in a fraction of the time that one would normally expect? And what if you could do it so that quality was improved and reliability and customer satisfaction increased with each new system? What if creating a new ship system was merely a matter of combining large, easily tailorable components under the auspices of a software architecture that was generic across the entire domain (in this case, of shipboard command and control systems)?

Is this a fantasy? No, it is not. It is the story of CelsiusTech Systems AB, a long-time European defense contractor. In the 1980s, CelsiusTech was confronted with the dilemma outlined above: They had to build two large command and control systems, each larger than anything they had attempted before, and they had barely enough resources to build one. Because necessity stimulates invention (and determination implements it), CelsiusTech realized that their only hope was to build both systems at once *using the same assets and resources*. And in a visionary stroke, CelsiusTech knew that their future lay in exploiting these assets for not only the first two systems but also for a whole family of products they hoped and expected would follow.

Software Product Lines

In short, CelsiusTech launched a *software product line*. A product line is a set of products that together address a particular market segment or fulfill a particular mission. Product lines promise to become the dominating production software paradigm of the new century. Product flexibility is the new anthem of the marketplace, and product lines fulfill the promise of tailor-made systems built specifically for the needs of particular customers or customer groups. What makes product lines succeed from the vendor's (developer's) point of view is that the commonalities shared by the products can be exploited to achieve economics of production.

Product lines are nothing new in manufacturing. Boeing builds one, so does Ford, IBM, and even McDonald's. Each of these exploits commonality in different ways. Boeing, for example, developed the 757 and 767 transports in tandem, and the parts lists of these two decidedly different aircraft overlap by about 60 percent. But *software* product lines based on interproduct commonality are a relatively new concept, and the community is discovering that this path to success contains more than its share of pitfalls.

The Software Engineering Institute (SEI) has a program to identify and promulgate the best practices for product-line production and help organizations negotiate the hurdles to which adopting a product-line approach will lead. The Product-Line Systems Program focuses on these essential technology areas for product-line production:

- Domain Engineering – Reveals the commonalities and variations among a set of products.
- Architecture – The foundation for a product line, it provides the framework into which tailorable components plug.
- Architecture-Based Development – The disciplined derivation or generation of product components (and once the components are ready, whole products) from the architectural skeleton.
- Reengineering – helps mine reusable assets from legacy assets.

The result is a technology infrastructure that can produce large custom systems quickly and reliably by checking out components from the asset repository, tailoring the components for their particular application (CelsiusTech uses compile-time parameters to instantiate different versions of a component), and beginning the integrate-and-test cycle as in normal system development.

Product Line Benefits

Once the product-line repository is established, consider what is saved each time a product is ordered.

- Requirements. Most of the requirements are common with earlier systems and therefore can be used. Requirements analysis is saved. Feasibility is assured.
- Architectural design. An architecture for a software system represents a large investment in time from the organization's most talented engineers. The quality goals for a system—performance, reliability, modifiability, etc.—are largely allowed or precluded once the architecture is in place. If the architecture is wrong, the system cannot be saved; however, for a new product, this most important design step is already done and need not be repeated.
- Components. The detailed (internal) designs for the architectural components are reused from system to system, as is the documentation of those designs. Data structures and algorithms are saved and need not be reinvented.
- Modeling and analysis. CelsiusTech reports that the real-time distributed headache associated with the kinds of systems they build (real-time distributed) has all but vanished. When they field a new product in their product line, they have extremely high confidence that the timing problems have been worked out, and the challenges associated with distributed computing—synchronization, network loading, and absence of deadlock—have been eliminated.
- Testing. Test plans, test processes, test cases, test data, test harnesses, and the communication paths required to report and fix problems are already available.
- Planning. Budgets and schedules can be reused from previous projects, and they are much more reliable.
- Processes. Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are in place, have been used before, and are robust, reliable, and responsive to the organization's special needs.
- People. Because of the commonality of the applications, personnel can be fluidly transferred among projects as required. Their expertise is applicable across the entire line.

Product lines enhance quality. Each new system takes advantage of all of the defect elimination in its forebearers; both developer and customer confidence rise with each new instantiation. The more complicated the system, the higher the payoff for solving the vexing performance, distribution, reliability, and other engineering issues only once for the entire family.

Clearly, product lines benefit the developing organization, but they also benefit acquirers of systems as well. Acquiring a family of related systems using a product-line acquisition approach (as opposed to acquiring each system separately and independently) clearly falls within the realm of DoD reuse initiatives and policies and promises to accrue significant benefits for the DoD, including

- Streamlining the acquisition process.
- Enjoying higher product quality.
- Lower acquisition cost.
- Simplified training.
- Reduced maintenance cost.

Organizational Maturity Needs

It takes a certain maturity in the developing organization to successfully field a product line. Technology is not the only barrier to successful product-line adoption. Experiences in the Product-Line Systems Program show that organization, process, and business issues are equally vital to master.

For instance, traditional organizational structures that have one business unit per product are generally not appropriate for product lines. Who will build and maintain the core reusable assets—the architecture, the reusable components, and so forth? If these assets are under the control of a business unit associated with one product or one large customer, the assets may evolve to serve that business unit, that product, and that customer to the exclusion of the others. On the other hand, to establish a separate business unit to work on the core assets but be divorced from working on individual products carries the danger that this unit will produce assets that emphasize beauty and elegance over practicality and utility. In either case, producing and managing the reusable assets means establishing processes to make the assets satisfy the needs of all of the business units that use them. This is a crucial role that requires staff skilled in abstraction, design, negotiation, and creative problem solving. The question of funding the core asset development is crucial.

Customer Management

Customer management becomes an important product-line function. Customers interact with a product-line organization in a different way. Marketers can no longer agree to anything customers want but must instead nudge customers to set their requirements so that they can be fulfilled by a version of the product line within the planned scope of variation.

Contrary to intuition, this often makes the customer much happier than before. Under the new paradigm, the marketer can point to specific requirements that would put the customer's new system outside the scope of the product line, which would increase the cost and delivery time, lower the system's reliability, and keep that customer out of a community of customers to which the vendor pays a lot of attention. Thus, the customer could clearly (and probably for the first time) see the real cost of those "special" requirements and make an informed decision about their real value. If the customer decides that a variant of the "standard" or product-line system will suffice, so much the better. If not, the customer can still order a system to satisfy particular requirements but with a better idea of where the risks may be hiding.

The customer community should not be underestimated. In CelsiusTech's case, their naval customers around the world banded together to form a users' group. They did this in their self-interest—to provide a forum in which they could jointly derive new requirements for their evolving systems and drive CelsiusTech to supply new systems more economically than they otherwise might. But it does not take much to realize how beneficial this is to CelsiusTech as well: Their customer base is jointly defining their next-generation products and effectively buying in to their approach, thus guaranteeing the vitality of their product line for years to come.

The users' group has a clear lesson for DoD acquisitions: It pays to collaborate (or at least communicate) when it comes to commissioning or purchasing similar systems.

Conclusion

Successful transition to product-line technology is thus a careful blend of technology, process, organization, and business factors improvement. The Product-Line Systems Program is attempting to codify these practices and understand how they vary with the type of organization involved and the kind of systems being built. Through a series of workshops, case studies, and collaborative engagements, SEI is helping to build a community of organizations interested in moving to a product-line approach for their software products.

We believe that product lines will be the predominant software paradigm at the beginning of the new century. The history of programming can be viewed as an upward spiral in which the abstractions manifested by components have grown larger and more application meaningful, with resulting increases in the reuse and applicability of those components. From subroutines in the 1960s to modules in the 1970s to objects in the 1980s to component-based systems in the 1990s, software product lines will perpetuate the upward spiral by accomplishing previously unheard-of levels of reuse from system to system.

If the pitfalls are successfully negotiated, the result is an enviable capacity to deliver extremely large systems on time and within budget.

For more information about the Product-Line Systems Program and its technology initiatives, visit SEI's Web page at http://www.sei.cmu.edu. You can download the full report that details the CelsiusTech product-line case study, which includes data about their dramatic results in time to market, levels of reuse, and required staffing. You also can read other product-line-related material, including the latest version of the SEI's Product-Line Practice Framework, a document that describes the essential practice areas of successful product-line development and acquisition. Contact the program manager, Linda Northrop, at lmn@sei.cmu.edu, for additional information. ◆

About the Author

Paul Clements is a senior member of the technical staff at Carnegie Mellon University's SEI. A graduate of the University of North Carolina and the University of Texas, he is a project leader in the SEI's Product-Line Systems Program. His work includes collaborating with organizations that are launching product-line efforts. He is a co-creator of the Software Architecture Analysis Method (SAAM), which allows organizations to evaluate architectures for fitness of purpose. He and others are working on an extension to SAAM, which will allow analysis of quality attribute trade-offs at the architectural level. He is co-author of *Software Architecture in Practice* (Addison-Wesley-Longman, 1998) and over three dozen papers and articles about software engineering.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
Voice: 512-453-1471
Fax: 412-268-5758
E-mail: clements@sei.cmu.edu

Managing (the Size of) Your Projects
A Project Management Look at Function Points

Carol A. Dekkers
Quality Plus Technologies, Inc.

This article is an introduction, as well as a refresher, to readers who need to update their knowledge about function points (FPs). It includes many of the concepts presented in my presentation, "Requirements Are (the Size of) the Problem," at the 1998 Software Technology Conference in Salt Lake City, Utah.

The use of FPs is becoming a widely supported method to create meaningful software metrics. In the past year, there has been a proliferation of software industry conferences with an increased emphasis on software measurement and FP-based metrics. Even tracks within traditionally non-measurement conferences now feature software measurement and FP presentations (the American Society for Quality's International Conference on Software Quality, the Quality Assurance Institute's International Quality Conference, the Software Technology Conference, and The MITRE Corporation's Software Engineering and Economics Conference, to name a few).

Prominent systems consultants are also increasingly vocal and supportive of FP-based metrics. For example, Howard Rubin based many of his workflow measurements on FPs (see *IT Metric Strategies*, December 1997), Capers Jones frequently highlights FPs as a basis for year 2000, productivity, and quality metrics; and Ed Yourdon discussed FPs in his presentation at the Applications of Software Measurement Conference (Atlanta, Ga., October 1997).

As president of the International Function Point Users Group (IFPUG), I appreciate the visibility generated for FPs by leading software practitioners, but I am disappointed that many professionals remain unfamiliar or misinformed about what functional size can and cannot do. There are even professionals who believe that by implementing FPs they can throw away all other software measures—they think FPs are *the* measurement program, but they are not.

FPs provide *one* additional project management tool, but they are not the entire toolkit. This article outlines the major components involved in FP counting, including counting examples. It ends with a look at the misconceptions about what FPs can and cannot do. (Other published articles such as "Demystifying Function Points – Understanding the Terminology" are available from the author.) This article also provides project managers with enough knowledge about FPs and their usage to make informed decisions about when and how to implement functional sizing on their projects.

What Are Function Points?

FPs measure the size of a software project's work *output* or work product rather than measure internal features such as lines of code (LOC). FPs evaluate the size of the functional user requirements that are supported or delivered by the software. In simplest terms, FPs measure *what* the software must do from an external, user perspective, irrespective of how the software is constructed. Similar to the way that a building's square measurement reflects the floor plan size, FPs reflect the size of the software's functional user requirements.

However, to know only the square foot size of a building is insufficient to manage a construction project. Obviously, the construction of a 20,000 square-foot airplane hangar will be different from a 20,000 square-foot office building. In the same manner, to know only the FP size of a system is insufficient to manage a system development project: A 2,000 FP client-server financial project will be quite different from a 2,000 FP aircraft avionics project.

I will carry the construction analogy one step further. To exclusively count the LOC (which many military departments do) is similar to counting the total number of construction pieces, e.g., 850,000 individual components that weigh 15,000 tons. This counting method is more indicative of construction methods than the functions (FPs) contained within a building. FPs are similar to summing up the square footage of functional items on a floor plan, e.g., the square footage of four sets of 10 x 15-foot restrooms, nine 12 x 20-foot meeting rooms, one 30 x 20-foot boiler room, two 10 x 40-foot staircases. FPs, like square feet, provide a normalized *size*, based on summing up the component functions that the resultant product must provide.

Because—no matter how large or how small—most software is developed to address user requirements, it can be measured in FPs. Some practitioners use the analogy of a "black box" to describe how FPs measure software independent of the inner workings of the software.

The process to calculate FPs is maintained by IFPUG and documented in its *Function Point Counting Practices Manual* (currently in release 4.0). Unlike LOC, FPs are independent of the physical implementation and languages used to develop the software, and they remain consistent no matter what development language or technique is used.

The fit between FPs and software development can be described analogously with square feet and construction (Table 1).

Why Use Function Points?

FPs provide an objective project size for use in estimating equations (together with other factors) or to normalize productivity or quality ratios. The value in using FPs lies in the ratios and normalized comparisons between ratios. Process improvements can be found when normalized ratios are compared and their underlying project attributes evaluated.

Metric	Construction Units of Measure	When Is It Important to Measure?	IT Units of Measure	When Is It Important to Measure?
Estimated Project Size	Square feet.	During floor plans stage.	FP.	During requirements or contract stage.
Unit Cost, Overall Cost	Cost per square foot, total cost.	During construction contract negotiation.	Cost per FP, cost.	Before go or no go development decision.
Estimated Work Effort	Man-months.	Throughout construction or whenever change occurs.	Hours or man-months.	Throughout development or whenever change occurs.
Size of Change Orders	Square feet, cost (impact).	Whenever change is identified.	FP, cost, or hours (impact).	Whenever change is identified.

Table 1. *Function points as a construction analogy.*

FPs provide a standard, normalized measure of the work product or *functional size* of software. Together with other measures, FP-based software metrics highlight process improvement opportunities and can increase estimating and prediction accuracy.

The Key to Counting Function Points: "Think Logical"

A fundamental feature of FP counting is that everything is counted from a *logical* user perspective, based on functional user requirements.[1] This is a paradigm shift for developers who are excellent at programming and physical configuration management. It does not matter to the functional size (FPs) whether it takes one thousand lines of COBOL code and eight subroutine calls or 100 lines of C++ code to perform a given business function; the FP count remains the same because the user function is the same.

Because excellent developers are akin to excellent plumbers involved in home construction, it takes a change in their mind-set to remove themselves from the physical implementation and look only at the floor plan. At this point, perhaps I have lost some developers who may think, "But the plumbing is important to keeping the house functioning. If we do not count other aspects of the software like the development language, we cannot accurately predict how long it will take to build." This assertion is absolutely correct, but functional size (FPs) is not the same as work effort. Here is the relationship:

Size (in FPs): An *independent* measure of the software's logical size. (Like the total room count and square footage of the finished building, which are constant regardless of construction methods.)

Work effort (in hours): A *dependent* measure of how long the software will take to develop, equal to a function of size, language, platform, skills, methods, team size, risks, and many other variables.

Productivity (in hours per FP): A *dependent* result, dependent on all the same factors as work effort. *Note that an independent variable (FP) divided by a dependent variable (hours) yields a dependent result.*

This means that just as the quality of the raw materials, piping configuration, and house layout affects the *work effort* it will take to plumb a house, so, too, will the language and other attributes affect software development time. However, regardless of *how* the house is designed and constructed, the functional size of the house stays the same. With software, the software *size* (in FPs) is independent of the language, skills, physical configuration, and other factors used in the development. When you use FPs, you are talking only about the software size.

What Gets Counted in Function Point Counting?

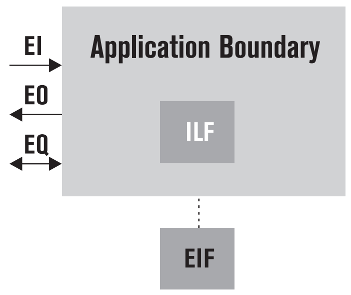
To count FPs, evaluate the following five *logical* components of the software based on the user requirements (Figure 1):[2]

- Internal logical files (ILFs) – Logical, persistent entities maintained by the software application.
- External interface files (EIFs) – Logical, persistent entities referenced only by this software application but which are maintained by another software application.
- External inputs (EIs) – Logical, elementary business processes that cross into the application boundary to maintain the data on an ILF or to ensure compliance with user business requirements, e.g., control data.
- External outputs (EOs) – Logical, elementary business processes that result in data leaving the application boundary to meet a user requirement, e.g., reports, data files, tapes, and screen alerts.
- External queries (EQs) – Logical, elementary business processes that consist of a data "trigger" followed by a retrieval of data that leaves the application boundary.

The five types of logical components counted are not the same as physical components. Discussion of ILFs, for example, does not refer to the physical files or data sets. ILF refers to the logical, persistent entities maintained through a standardized function of the software—they are the stand-alone, logical entities that typically would appear on an entity relationship diagram. For example, in a human resources application, an associate or employee entity would be maintained. This entity would be counted as an ILF.

Another illustration of counting logical components is when referring to EIs, which are the logical, elementary business processes that maintain the data on an ILF or that control processing.

Figure 1. *IFPUG function point components in relation to the application.*



Making Adjusted FP Counts

Without getting into *IFPUG Function Point Counting Practices Manual* specifics of how to rate components as low, average, or high complexity, the following illustrates how to arrive at the unadjusted FP count for an application or development project. Following are the functional user requirements.

- Create an application to store and maintain employee records consisting of the following data fields: name, number, rank, street address, city, state, ZIP code, date of birth, telephone number, office assigned, and date the employee data was last modified.
- The application must provide a means to add new employees, update employee information, terminate employees, and merge two employee records.
- The application must provide a weekly report on paper that lists which employees' information has changed during the past week.
- The application must provide a means to browse employee data.
- No data outside the application is referenced, and all data validation edits are done using hard coded (not modifiable) data.

The FP components to be counted based on the above include

- **One ILF** for the employee data because it is a persistent logical entity maintained by the application. Based on an evaluation of the data elements and logical record types (contained in the counting practices manual), this ILF would be categorized low and be worth seven FPs.
- **Four EI processes:** One EI each for add employee, update employee, terminate employee, and merge employee records. Assuming each one is of low complexity (each requires only one logical entity and requires fewer than 16 data elements), each EI would be worth three FPs for a total of 12 FPs.
- **One EO process:** The weekly report would be categorized as an external output and typically would consist of fewer than 20 data elements and require only the employee logical file. Based on the counting rules, this external output would be classified low and be worth four FPs.
- **One EQ process.** The process to browse the employee data would be classified as an EQ. Based on the number of data elements (fewer than 20) and the number of logical files accessed (the employee ILF), this EQ would be classified low and be worth three FPs.

Total components from the above four points: 26 unadjusted FPs. The final step would be to determine the value adjustment factor based on the user business constraints evaluated per the *Function Point Counting Practices Manual.* Guidelines are provided in the manual to help FP practitioners properly evaluate the adjustment factor.

The logical business process of adding an associate would be one user function; therefore, in function point counting, you would count one EI. The size in FPs for this one EI would be the same regardless of how we physically implemented it because in every implementation, it performs one logical user function. For example, the count for "add associate" is the same regardless of the number of screens, keystrokes, batch programs or pop-up data windows needed to complete the process.

Table 2. *Unadjusted FP values by component.*

Function Type	Low	Average	High
EI	x 3	x 4	x 6
EO	x 4	x 5	x 7
EQ	x 3	x 4	x 6
ILF	x 7	x 10	x 15
EIF	x 5	x 7	x 10

What Is Involved in Function Point Counting?

The basic steps[3] involved in function point counting include

- Determine type of count, e.g., new development project, application or base count, or enhancement project count.
- Identify the application boundary, i.e., what functions must the software perform? This creates a context diagram for the application or project.
- Count the data function types.
  - ILFs – logical data groups maintained within the application boundary.
  - EIFs – used only for reference by the application.
- Count the transactional function types.
  - EIs – data entry processes and controlled inputs.
  - EOs – e.g., reports.
  - EQs – e.g., question-and-answer pair that results in data retrieval.
- Evaluate the complexity of each of the five function types identified above. IFPUG provides several simple matrices to determine whether a function is low, average, or high, based on data element types (user recognizable, non-recursive data fields), record element types (subsets of user-recognizable data), and file types referenced (number of logical data groupings required to complete a process). Table 2 summarizes the number of FPs assigned to each function type. Following the IFPUG guidelines, count and rate all the identified functions and add the FPs together. The resulting number is the *unadjusted FP count.*
- Determine the value adjustment factor (VAF), which reflects the user problem complexity for the developed software. The VAF is calculated via an equation (VAF = 0.65 + (sum of GSCs x .01) and the evaluation of the following 14 general system characteristics (GSCs). Specific evaluation guidelines for the following GSCs are provided in the IFPUG *Function Point Counting Practices Manual.*
  - Data Communication.
  - Distributed Data Processing.
  - Performance.
  - Heavily Used Configuration.
  - Transaction Rate.
  - On-Line Data Entry.
  - End-User Efficiency.
  - On-Line Update.
  - Complex Processing.
  - Reusability.
  - Installation Ease.
  - Operational Ease.
  - Multiple Sites.
  - Facilitate Change.
- Calculate the final adjusted FP count (adjusted function count = unadjusted FP count x VAF).

The Logical Boundary

One of the first steps of counting FPs is to identify the logical boundary around a software application. This "boundary" separates the software from the user domain. (Users can be people, things, other software applications, departments, other organizations, etc.) As such, the software may span several platforms and include both batch and on-line processes. The boundary is not drawn around the software in terms of how the system is implemented but rather in terms of how an experienced user would view the software. This means that a single application boundary can encompass several hardware platforms, e.g., both mainframe and PC hardware used to provide an accounts receivable application would be included within the application boundary.

Where Do Function Points Fit In?

Once the adjusted FP count for a project or application has been created, it becomes the size of the work product. Just as the total functional size of a house does not equal the speed at which a house can be built or its construction time, the FP size does not equal productivity or work effort. FPs measure the size of *what* the software does, rather than *how* it is developed and implemented. This means that given a common set of logical user requirements, the FP size of the software will be the same whether it is developed using COBOL or DB2 or using rapid application development or structured development methods.

Where Can I Learn More About Function Points?

If you are going to get serious about software measurement or FPs or just want further information on how to get started with a measurement program, contact IFPUG or me. Incorporated in 1986, IFPUG is a not-for-profit users' group that has become a leader in establishing and publishing function point-related documents, including the *Function Point Counting Practices Manual*, the *Guidelines to Software Measurement* (currently in release 1.1), *Function Points as Assets* guide, and several detailed FP case studies. IFPUG remains a volunteer organization (with a small, paid administrative staff), is active in International Organization for Standardization (ISO) standards, sponsors conferences and workshops, and certifies FP training, counters (certified function point specialists), and FP software. Currently, paid membership represents over 30 countries worldwide. ◆

About the Author

Carol A. Dekkers is president of Quality Plus Technologies, Inc., a management consulting firm specializing in training and consulting in function points, software metrics, requirements, and estimation process improvement. She is president of the IFPUG board of directors

and is a project editor within the ISO functional size measurement workgroup (ISO/IEC/JTC1/SC7 WG12). She is a frequent presenter and trainer at both U.S. and international quality and measurement conferences and is credentialed as a certified management consultant, a certified function point specialist, a professional engineer (Canada), and an Information Systems professional.

Quality Plus Technologies, Inc.
8430 Egret Lane
Seminole, FL 33776
Voice: 727-393-6048
Fax: 727-393-8732
E-mail: dekkers@qualityplustech.com
Internet: http://www.qualityplustech.com

Types of Function Point Counts

The full details of FP counting procedure is contained in the IFPUG *Function Point Counting Practices Manual*, Version 4.0. There are two major types of FP counts:

- **Application or base FP count**: This count is the size of an installed base application. (Think of it in terms of total square feet of an existing house). The base size in FP is a point-in-time snapshot of the current size of an application. This number is useful whenever comparisons are required between different applications, e.g., defects divided by base FPs.
- **Project or enhancement FP count:** This count reflects the size of the functional "area touched" by an enhancement project. An enhancement project count is the result of summing the new functions added in the project plus the functions removed from the application by a project plus the functions changed by the project. (Think of this count in terms of a renovation project where the square foot of the project equals the sum of the area of a new living room, a removed bathroom, and a remodeled kitchen). This count is useful in project-based metrics, e.g., relative cost in dollars divided by development FP.

As the project is finished, the application or base FP count must be updated by the net, i.e., new minus removed plus the net difference in the changed functions.

Notes

1. If you do not have "functional user requirements," does that mean you cannot count FPs? There are always requirements, although they may not be fully articulated by users or documented in a clear and complete fashion. In the early stages of software development, you may have to estimate the requirements or make assumptions about the user requirements and subsequently base your count on those assumptions. (See "Requirements Are [the Size of] the Problem," *ITMetric Strategies*, March 1998, which further explores the topic.)
2. The components listed are taken from the IFPUG *Function Point Counting Practices Manual*, Version 4.0, February 1994. The explanatory text in italics below each component is my wording to describe each component. For further information or to obtain the manual, contact the IFPUG administrative office in Westerville, Ohio at 614-895-7130 or visit the Web site at http://www.ifpug.org.
3. These steps condense the full details of function point counting included in the *Function Point Counting Practices Manual*, Version 4.0. Additionally, there are full case studies of FP counts, done at differing phases of application development, that can also be ordered through the IFPUG office.

The Upside of Y2K

John B. Hubbs
AverStar, Inc.

Despite the potential catastrophes of failure, a long-term view of solving the Y2K problem provides many positive benefits. The opportunities should be seized for more objective communications within the commands and for improved processes within our software shops.

Information technology (IT) industry pundits and the press predict all types of calamities to befall humanity because of the year 2000 (Y2K) crisis. Some of the worst-case scenarios may manifest themselves, and technology's reputation will take a beating from those most injured. Despite the near-term downside of Y2K problems, the long-term view indicates many beneficial aspects to "the crisis."

The major problem IT faces is described in the popular press as a "bug" or a "glitch." These terms are commonly used by practitioners for relatively minor problems that can be solved with minimal effort. The truth is that information technologists created the Y2K problem by conscious decisions; that is, we *designed* a crisis. But crises frequently have positive aspects when viewed from a long-range perspective.

Other articles have addressed the technical issues Department of Defense commanders and information technologists face. This article details the positive aspects of solving Y2K problems. When we look back on this era 10 or 20 years from now, we will appreciate what the millennium has done to transform our industry.

The common thread of the upside of Y2K is a focus on process. Whether a five-step Y2K process or the Capability Maturity Model (CMM), many software engineering processes will be enhanced by the year 2001. The positive aspects of Y2K can be grouped into four major areas:

- Improved structure.
- Heightened software awareness.
- Enforced accountability.
- Industrial maturity.

Improved Structure

After an inventory of systems (also a long-term benefit), the next step in addressing an organization's Y2K situation is the decision to retire applications not worth renovating. Many of these older systems should have been retired many years ago. But we are reluctant to pull the plug on these stalwart applications. Building in the "millennium bug" when storage was a premium created unplanned obsolescence. After 1999, we will no longer need to maintain systems for which the functions will be delivered with more contemporary software. The cost to maintain applications will be reduced through

- fewer lines of code to support.
- a refreshed understanding of the remaining programs.

Organizations that started their Y2K activities a year or two ago took advantage of the opportunity to thoroughly review their systems. One health-care claims processing software shop that carefully examined all its applications in 1994 reported a reduction of more than 10 percent in lines of code compiled upon completion of their Y2K project. Organizations that have completed a reengineering with their Y2K renovation also report less cycle time to execute the more efficient applications.

Shops that started later had to employ fully automated techniques without a comprehensive review to optimize systems before renovation. These shops see little code reduction and, when using windowing techniques, may experience a small increase. Regardless of the amount of reengineering, additional benefits to the renovation process include expansion of the regression testing suite.

On the hardware side, many commands experienced a proliferation of platforms over the past several years. Improved infrastructure provides many of these shops with the opportunity to standardize on a processor base that should consolidate future maintenance costs. As commanders stabilize their platforms, they will also be using the Y2K process as an opportunity to reduce the multiplicity of operating systems and to benefit from economies of scale.

Heightened Software Awareness

Perhaps one of the most profound and subtle positive effects of the Y2K problem is the new level of dialog between senior managers and information technologists. Over the past 25 years, data processing has grown from an ancillary function to a ubiquitous necessity of modern life. Many managers in government and business, however, continue to see IT as a "back room" support function. The abstract nature of software once facilitated a manager's ability to relegate our profession to a support role.

With the millennium approaching, managers are more acutely aware of the role of software to the continued successful operation of the organization. Senior commanders and boards of directors now are aware of software risk management, project staffing, task status, metrics, and contingency planning. Although upper-level management need not be involved in the details of IT over the long range, their concerns about the Y2K project bodes well for the future of our profession. Their increased awareness of the processes required to create and maintain software should reduce the number of arbitrary, uninformed decisions in the future and foster better communications with technologists. Programmers at Johns Hopkins University report that their board of directors has a Y2K oversight committee that tracks progress and issues on a regular basis. Nontechnical managers at all levels will have a better appreciation for software and the extensive testing required to get it right.

In addition to systems "owned" by a command, interfaces with other applications have increased with advances in telecommunications. With each command serving its own needs, we have unintentionally woven a complex web of interfacing systems. The Y2K issue forces us to realize that the interfaces are as important as the home-grown systems. As information technology practitioners, we should not let this opportunity pass us by to inform our managers.

Enforced Accountability

Partly because our work is viewed as a "black art," we have been allowed slippages and nonperformance that would not be acceptable in other parts of the military. Software project failures and cost increases are not as publicized in the popular press as much as hardware fiascoes. Software projects have been given more time and money despite

- The shortage of documented requirements.
- Demands to meet an often arbitrary deadline.
- Confusion about the process.
- A changing technologic base.
- Frequent reorganizations.

Yet, this "new industry" has fostered unprecedented growth in an economy that was all but given up as lifeless 20 years ago when we feared being overrun by the Japanese. Most of this success was achieved by heroes who sacrificed a good share of their personal life to bring order out of chaos.

With an unchangeable due date that cannot be missed, we will be held responsible for our performance. The processes that must be enforced to be Y2K successful have grabbed the attention of senior officers (assuming they know they need a disciplined software process). Failure is not an option without bringing down the enterprise—be it a command, a business, or a government agency. Technologists will be fully accountable for Y2K success or failure.

Industrial Maturity

Any new industry experiences growing pains. The opportunities of the industrial revolution at the end of the last century gave rise to labor abuses and monopoly powers. Through painful experience, a free society regulated itself to create a positive, constructive environment in which the then new technologies could flourish. Electricity was as misunderstood and ubiquitous at the turn of the last century as software is at the turn of the millennium.

Thus, the millennium change is our epiphany—the hidden art of software engineering will come out of obscurity to be seen and to serve in a more open and visible manner. This coming out will be accompanied by a renewed focus on the processes by which software is conceived and delivered. Those organizations that handle Y2K successfully will be seen as more progressive and able to deal with future challenges. Those who fail will join the manufacturers of muzzleloaders or will be relegated to forever catching up until they are overtaken by new market or regulatory forces.

The rapid growth of information technologies has rendered established techniques out of date. The cry of the hands-on technologist is the need to "maintain my skills." Everyone wants to work on new development projects using the latest technology. As soon as a new technique catches on, the "old" skills are no longer desirable.

Only a few years ago, COBOL programmers were concerned about two aspects of their jobs: being relegated to maintenance tasks and not staying current with the latest technology. Today, these same people are being recalled from retirement with attractive salaries to perform maintenance on the old mission-critical systems. Skills in COBOL and the older information technologies enjoy a new-found respect that will last many years.

Our industry has grown so rapidly that we have not taken the time to step back and see what we have wrought. Individual commands focus on their system needs and develop interfaces with other systems for even more efficiencies. A similar approach to address Y2K is being taken. Each organization has looked at its system inventory and (I hope) taken appropriate action. Only late in the game has it become apparent that the Y2K issues of our partners (suppliers of data as well as software) are as important as our internal problems. The worst-case scenarios of a global data meltdown are based on an understanding of the intricate, interlocking webs of applications that were built long before the Advanced Research Project Agency conceived the Internet. If any of these scenarios become real, we will fully realize for the first time how pervasive software has become.

Another subtle change in attitude centers on motivation. Nature's primary motivators are desire and fear. As society exploited computational skills, management and practitioners were motivated mostly by desires—to beat the competition (or enemy), to code a more sophisticated routine, or to engineer another massive system. The motivator for the Y2K problem, however, is fear. For commands actively working on Y2K problems, management should be fully involved—failure is not an option. Fear of failure is more tangible now than it ever has been in this industry. The new motivator is another basis for maturation of our profession.

Conclusion

The Y2K issues and fears we face today are substantial. When we look back on the Y2K problem in 2015, we will see it as a blessing that we designed into an infant industry. Many more positives than those outlined here will be visible from the perspective of hindsight.

With an accurate inventory and improved regression test suite, programmers will be able to maintain systems more efficiently than in the past. As a result of the intense focus on Y2K projects, senior commanders and executives will have a better appreciation of the configuration management and quality assurance processes necessary to develop and support systems. With better processes, project managers can estimate costs and schedules more accurately and be more accountable for the results. Thus, a more mature industry that has survived a dilemma of its own making may be seen as a valuable asset.

As information technology survives the millennium, the perception of our industry will move from that of an art to a science. As we struggle with our Y2K problems through the next few years, let us not lose sight of the many potentially beneficial aspects that improved processes will provide us. In the future, commanders and practitioners alike will be able to focus more on technical issues than process problems. ◆

About the Author

John B. Hubbs designed the Civilian Health and Medical Program for the Uniformed Services health-care claims processing system deployed with the 7th Army in Heidelberg, Germany. While supporting the Social Security Administration, he directed the contractor's CMM-based process improvements and lead the design for a major development project that received Vice President Gore's National Performance Review Award in June 1997. He currently performs independent verification and validation for the Health Care Financing Administration.

AverStar, Inc.
3120 Timanus Lane
Baltimore, MD 21244
Voice: 410-944-3190
Fax: 410-944-3868
E-mail: jhubbs@averstar.com
Internet: http://www.averstar.com

Coming Events

Call for Papers: 1999 Acquisition Research Symposium (ARS)

Dates: June 21-23, 1999
Location: Doubletree Hotel, Rockville, Md.
Sponsor: Deputy Undersecretary of Defense for Acquisition Reform.
Co-hosts: Defense Systems Management College (DSMC) and the National Contract Management Association, Washington, D.C. chapter.
Theme: "Acquisition Reform – A Revolution in Business Affairs"
Topics: Acquisition Reform Successes and Lessons Learned, Business Process Reengineering and Benchmarking, Commercial Applications in Government, Competitive Acquisition Strategies, Cost and Resource Management, Federal Acquisition and the Political Process, Industrial Base/Civil/Military Integration, International Acquisition Issues, Leveraging Technology in Acquisition, Management Decision and Information Support Tools, Organization and Cultural Change, Outsourcing and Privatization.
Three camera-ready copies of paper due: Feb. 26, 1999 (obtain guidelines from Joan L. Sable)
Submit to: Joan L. Sable, DSMC Program Chairwoman, ARS 99, 9820 Belvoir Road, Suite 3, Fort Belvoir, VA 22060-5565
Voice: 703-805-5406 DSN 655-5406
E-mail: ars99@dsmc.dsm.mil

"Making the Move to COCOMO II" Seminars

Dates: March, May, and September 1999
Locations: Los Angeles, Calif., San Francisco, Calif., Dallas, Texas, Chicago, Ill., Boston, Mass., Washington, D.C., depending on date.
Instructor: Donald J. Reifer
Voice: 310-530-4493
Fax: 310-530-4297
E-mail: info@reifer.com
Internet: http://www.reifer.com

1999 IEEE Aerospace Conference

Dates: March 6-13, 1999
Location: Snowmass at Aspen, Colo.
Sponsor: Aerospace and Electronics Systems Society of the Institute of Electrical and Electronics Engineers (IEEE).
Topic: The internationally attended IEEE Aerospace Conference is organized to promote interdisciplinary understanding of aerospace systems, their underlying science and technology, and their applications to government and commercial endeavors.
Contact: Mike Johnson or Beth Leitereg
Voice: 702-784-6485
Fax: 702-787-1342
E-mail: jmj@ee.unr.edu
Registration: registration@aeroconf.org
Internet: http://www.aeroconf.org

Call for Papers and Workshops: Ninth International Conference on Software Quality

Dates: Oct. 4-6, 1999
Location: Royal Sonesta Hotel, Cambridge, Mass.
Sponsor: American Society for Quality Software Division
Suggested Topics: Software Quality Management, Software Processes, Software Project Management, Software Metrics, Measurement and Analytical Methods, Software Inspection, Testing, Verification and Validation, Software Audits, Software Configuration Management, Standards, Quality Philosophies and Principles, Organizational and Interpersonal Techniques, Problem-Solving Tools and Processes, and Professional Conduct and Ethics.
Abstracts, proposals for workshops, and panel sessions due: March 8, 1999
Contact: John Pustaver, Conference Chairman, 4 Kendall Road, Sudbury, MA 01776
E-mail: pustaver@swquality.com

It's Time to Register for the Eleventh Annual Software Technology Conference

The Eleventh Annual Software Technology Conference (STC '99) will be held in Salt Lake City, Utah, May 2-6, 1999.

The U.S. Air Force, Army, Navy, Marine Corps, and the Defense Information Systems Agency (DISA) have again joined forces to co-sponsor STC '99, the premier software technology conference in the Department of Defense (DoD). Once again, Utah State University Conference Services is the conference nonfederal co-sponsor.

The government co-sponsors are Lt. Gen. David J. Kelley (DISA), Lt. Gen. William Campbell (U.S. Army), Dr. Helmut Hellwig (U.S. Air Force), Rear Adm. Kenneth D. Slaght (U.S. Navy), and Brig. Gen. Robert Shea (U.S. Marine Corps).

The theme for STC '99, "Software and Systems for the Next Millennium," is shaping up to be a transition conference reflecting the convergence of the DoD's tactical and nontactical information systems, processes, people, and policy in support of our war fighters. This theme reflects the broader role of software within the domain of knowledge sharing. Going beyond mere transmission of data elements, knowledge sharing identifies the need for contextual exchange of situational awareness within the dispersed, rapid-paced battle space in which modern U.S., allied, and coalition forces operate. STC '99 showcases more than just a "software technology conference"; it sharpens the focus of the ubiquitous role of software, information technology, and information warriors as they support our military capabilities.

We anticipate over 3,500 participants from the military services, government agencies, contractors, industry, and academia.

The general session will be held on Monday afternoon. In addition to the general session, the co-sponsors have agreed to host a question-and-answer general session on Tuesday morning, when they will address questions from conference participants. Conference attendees are encouraged to turn in their questions to conference management prior to the conference or to the on-site control room on Monday of the conference week. This is a great chance to get answers from our senior leaders on important issues.

Over 600 excellent abstracts were submitted this year by potential speakers, which made choosing a few more than 100 speakers extremely difficult. Participants will be greatly pleased with the selection of speakers and topics.

There will also be a special intelligence meeting held in conjunction with the conference on Wednesday. Top Secret SI/TK clearances will be necessary for this one-day track. Details are in the registration brochure.

Our exhibit area has grown to 324 vendor booths. At press time, limited space is still available. Registration information is in the registration brochure. For current vendor information, please check the STC '99 Web site at http://www.stc-online.org. Space rental rate is $1,175 per 10-foot by 10-foot booth. Late registration received after Feb. 12, 1999, should space be available, will rent for $1,275 per booth. A copy of the exhibitor registration brochure with a full layout of the exhibition area is available on the Internet at http://www.stc-online.org. For more information concerning the exhibition, E-mail a request to stcexhibits@ext.usu.edu, use fax-on-demand at 435-797-2358, or call 435-797-0047.

One of the greatest benefits of STC is that it provides excellent networking opportunities. Side meetings and Birds-of-a-Feather meetings are already being scheduled. If you are interested in reserving a time for one of these meetings, please call us at 801-777-9828, and we will be glad to schedule a meeting for you.

Hotel guest room reservations are being taken through the Salt Lake Convention and Visitors Bureau (SLCVB). To reserve your hotel guest room, fill out the form in your registration brochure or call fax-on-demand 435-797-2358 for a housing reservation form. As soon as possible, fax it to the SLCVB Housing Bureau at 801-355-0250. Government-rate rooms are filled quickly. Buses will be provided to help with transportation between the conference center and city-center hotels during conference hours. Be sure to read the instructions on the housing form closely.

The conference fee structure for STC '99 is

Discounted registration fee paid by March 26, 1999	
Active Duty Military/Government	$465*
Business/Industry/Other	$585
Regular registration fee paid after March 26, 1999	
Active Duty Military/Government	$515*
Business/Industry/Other	$635

* Military rank (active duty) or government GS rating or equivalent is required to qualify for this rate.

The official STC '99 registration brochure was mailed in early January. We have made it easier to register early this year. Send in your registration forms with your credit card number *now*, and it will not be charged until March 25, 1999. You no longer have to wait until May to register.

All CROSSTALK subscribers are on our mailing list. If you borrow CROSSTALK, please contact us to be added to our mailing list.

You may use our Web site at http://www.stc-online.org for further information about STC '99.

If we can be of further assistance, please call or E-mail. This is one conference that you do not want to miss. We will see you in May!

Dana Dovenbarger, Conference Manager
Lynne Wade, Assistant Conference Manager
Software Technology Support Center
OO-ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205
Voice: 801-777-7411 DSN 777-7411
Voice: 801-777-9828 DSN 777-9828
Fax: 801-775-4932 DSN 775-4932
E-mail: dovenbad@software.hill.af.mil
wadel@software.hill.af.mil

Legacy of the Information Age

Every era of human history is designated by an "age," such as the Iron Age, the Industrial Age, the Space Age, and the Age of Flared Pants and Shirt Lapels Big Enough to Shade Municipal Stadiums. How will our age be remembered by future historians? What earth-changing creations of our era will take place alongside the wheel, the printing press, the factory, the telephone, and stone-washed jeans?

The namesake invention of our era should be so pervasive that it affects the way we work, play, and think. And the winner becomes painfully self-evident to me as I type this column on my *word processor* in preparation to *E-mail* it to our *Webmaster:* We'll be remembered as the Age of Multilevel Marketing Schemes. I tell you, it's the wave of the future, taking advantage of the vast network of family and friends you already have and providing almost limitless opportunities to anger and alienate them—although they'll someday thank you when their hard work blossoms into never-ending wealth for some schmuck two levels above you.

However, my grandparents have been around for nearly a century, and they believe the biggest invention of their lifetimes—even ahead of television, the airplane, and "The Clapper"—has been the computer. They may have a point. It's hard to appreciate what an impact they've had on society (computers, not my grandparents) without stepping back and imagining how key historical periods would have been altered if the players had had the benefits of computers:

Martin Luther: I hath thrice E-mailed thee without response. Art thou hastily printing my Biblical translation?

Johannes Gutenberg: Nay, I art ensnared in a game of Mule Simulator II. Didst thou send thy file?

Luther: Thou hast not seen it? I hath sent it erenow a fortnight!

Gutenberg: Perchance my mail server doth have the palsy anew. Lo, a half moon aforehence I hath received a scrip that I couldst not readeth, for verily it wert not properly unencoded.

Luther: Must I nail the scrip to thy door? I shall sendeth a new copyeth so thou canst printeth in hasteth!

Gutenberg: Have not a heifer! Lo, I am bound to take respite from my toils at present, for my press art incompatible with "Parchment 1495." I awaiteth a hardware upgrade and a beta of "Parchment 1500."

Luther: The Reformation canst not wait! When wilt it arrive?

Gutenberg: Perchance decades. After all, I dieth in 1468.

Luther: Forsooth! Thou dost yield the ghost 15 years before my very birth. 'Tis truly vexing we chance to speak!

Gutenberg: Yea. And shouldn't we be speaking German?

So you can see why we owe such a debt of gratitude to computers. For example, the above exchange demonstrates how computers enhance our effectiveness in one of the most critical aspects of business: having a believable excuse for not getting stuff done.

The old technology just wasn't working right. Receptionists always blather to annoying clients that they left the phone message "right on your chair." The postal service has become too reliable for anyone to believe "the check is in the mail." So we were long overdue for the powerful alibis computers provide:

"Our E-mail is down."

"My voice mail system erased your message."

"My dog ate my network password."

"Our E-mail ate your network."

"Your voice mail erased my dog."

We all experience these problems from time to time, so people believe us. And with no humans in the electronic loop, these excuses are tough to disprove.

I'd share other ways computers improve our lives, but unfortunately, a virus scrambled them on my hard drive—which demonstrates the need for *reliable* computers you can buy from people you *trust*. There's a huge untapped market for this type of buying—the path to financial independence! If you get in on the ground floor now by selling reasonably priced computer products to your family and friends (and get them to sell to their friends, and so on), I guarantee that you'll soon have nothing to do but kick back on your new yacht and tell your forklift driver where to deposit the morning's usual incoming bale of $100 bills.

Send $100 in care of this journal, and I'll forward information on this exciting opportunity just as soon as I cash your check—that is, unless my E-mail server gets the palsy. If you experience delays, just leave me a voice mail. – Lorin May

Got an idea for BACKTALK? Send an E-mail to backtalk@stsc1.hill.af.mil

CROSSTALK

Sponsor	*Lt. Col. Joe Jarzombek* 801-777-2435 DSN 777-2435 jarzombj@software.hill.af.mil
Publisher	*Reuel S. Alder* 801-777-2550 DSN 777-2550 publisher@stsc1.hill.af.mil
Managing Editor	*Tracy Stauder* 801-775-5746 DSN 775-5746 managing_editor@stsc1.hill.af.mil
Senior Editor	*Sandi Gaskin* 801-777-9722 DSN 777-9722 senior_editor@stsc1.hill.af.mil
Graphics and Design	*Kent Hepworth* 801-775-5798 graphics@stsc1.hill.af.mil
Associate Editor	*Lorin J. May* 801-775-5799 backtalk@stsc1.hill.af.mil
Editorial Assistant	*Bonnie May* 801-777-8045 editorial_assistant@stsc1.hill.af.mil
Features Coordinator	*Denise Sagel* 801-775-5555 features@stsc1.hill.af.mil
Customer Service	801-775-5555 custserv@software.hill.af.mil
Fax	801-777-8069 DSN 777-8069
STSC On-Line	http://www.stsc.hill.af.mil
CROSSTALK On-Line	http://www.stsc.hill.af.mil/Crosstalk/crostalk.html
ESIP On-Line	http://www.esip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address:

Ogden ALC/TISE
7278 Fourth Street
Hill AFB, UT 84056-5205

E-mail:	custserv@software.hill.af.mil
Voice:	801-775-5555
Fax:	801-777-8069 DSN 777-8069

Editorial Matters: Correspondence concerning *Letters to the Editor* or other editorial matters should be sent to the same address listed above to the attention of *CROSSTALK Editor* or send directly to the senior editor via the E-mail address also listed above.

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 *Guidelines for CROSSTALK Authors*, available upon request. We do not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

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

Trademarks and Endorsements: All product names referenced in this issue are trademarks of their companies. The mention of a product or business in CROSSTALK does not constitute an endorsement by the Software Technology Support Center (STSC), the Department of Defense, or any other government agency. The opinions expressed represent the viewpoints of the authors and are not necessarily those of the Department of Defense.

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

STSC On-Line Services: STSC On-Line Services can be reached on the Internet. World Wide Web access is at http://www.stsc.hill.af.mil. Call 801-777-7026 or DSN 777-7026 for assistance, or E-mail to schreifr@software.hill.af.mil.

Publications Available: The STSC provides various publications at no charge to the defense software community. Fill out the Request for STSC Services card in the center of this issue and mail or fax it to us. If the card is missing, call Customer Service at the numbers shown above, and we will send you a form or take your request by phone. The STSC sometimes has extra paper copies of back issues of CROSSTALK free of charge. If you would like a copy of the printed edition of this or another issue of CROSSTALK, or would like to subscribe, please contact the customer service address listed above.

The ***Software Technology Support Center*** was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies that will improve the quality of their software products, their efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery. CROSSTALK is assembled, printed, and distributed by the Defense Automated Printing Service, Hill AFB, UT 84056. CROSSTALK is distributed without charge to individuals actively involved in the defense software development process.