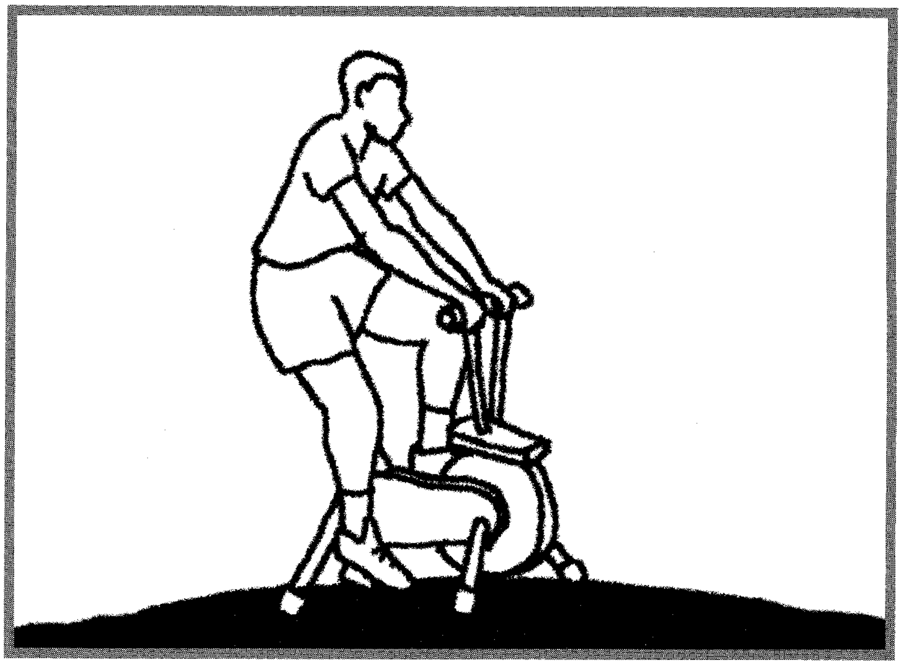
feature

Using A Defined and Measured Personal Software Process

Improved software processes lead to improved product quality. The Personal Software Process is a framework of techniques to help engineers improve their performance — and that of their organizations — through a step-by-step, disciplined approach to measuring and analyzing their work. This article explains how the PSP is taught and how it applies to different software-engineering tasks. The author reports some promising early results.



WATTS S. HUMPHREY
Software Engineering Institute

Fewer code defects, better estimating and planning, enhanced productivity — software engineers can enjoy these benefits by learning and using the disciplines of the Personal Software Process. As a learning vehicle for introducing process concepts, the PSP framework gives engineers measurement and analysis tools to help them understand their own skills and improve personal performance. Moreover, the PSP gives engineers the process understanding they need to help improve organizational performance. Up to a point, process improvement can be driven by senior management and process staffs. Beyond Level 3 of the Software Engineering Institute's Capability Maturity Model, however, improvement requires that engineers apply process principles on an individual basis.[1]

In fact, it was because of the difficulties small engineering groups had in applying CMM principles that I developed the PSP. Large and small organizations alike can benefit from CMM practices, and I focused the original PSP research on demonstrating how individuals and small teams could apply process-improvement methods.

In this article, I describe the PSP and experiences with teaching it to date. Thus far, the PSP is introduced in a one-semester graduate-level course where engineers develop 10

TABLE 1
PSP EXERCISES

Program Number	Brief Description
1A	Using a linked list, write a program to calculate the mean and standard deviation of a set of data.
2A	Write a program to count program LOC.
3A	Enhance program 2A to count total program LOC and LOC of functions or objects.
4A	Using a linked list, write a program to calculate the linear regression parameters (straight line fit).
5A	Write a program to perform a numerical integration.
6A	Enhance program 4A to calculate the linear regression parameters and the prediction interval.
7A	Using a linked list, write a program to calculate the correlation of two sets of data.
8A	Write a program to sort a linked list.
9A	Using a linked list, write a program to do a chi-squared test for a normal distribution.
10A	Using a linked list, write a program to calculate the three-parameter multiple regression parameters and the prediction interval.
1B	Write a program to store and retrieve numbers in a file.
2B	Enhance program 1B to modify records in a file.
3B	Enhance program 2B to handle common user errors.
4B	Enhance program 3B to handle further user error types.
5B	Enhance program 4B to handle arrays of real numbers.
6B	Enhance program 5B to calculate the linear regression parameters from a file.
7B	Enhance program 6B to calculate the linear regression parameters and the prediction interval.
8B	Enhance program 5B to sort a file.
9B	Write a program to do a chi-squared test for a normal distribution from data stored in a file.
Reports	
R1	LOC counting standard: Count logical LOC in the language you use to develop the PSP exercises.
R2	Coding standard: Provide one logical LOC per physical LOC.
R3	Defect analysis report: Analyze the defects for programs 1A through 3A.
R4	Midterm analysis report of process improvement.
R5	Final report of process and quality improvement and lessons learned.

module-sized programs and write five analysis reports. Early results are encouraging — while individual performance varies widely, data on 104 students and engineers show reductions of 58 percent in the average number of defects injected (and found in development) per 1,000 lines of code (KLOC), and reductions of 71.9 percent in the average number of defects per KLOC found in test. Estimating and planning accuracy are also improved, as is productivity — the average improvement in LOC developed per hour is 20.8 percent.

You can apply PSP principles to almost any software-engineering task because its structure is simple and independent of technology — it prescribes no specific languages, tools, or design methods.

PSP OVERVIEW

A software process is a sequence of steps required to develop or maintain software. The PSP is supported with a textbook and an introductory course.[2] It uses a family of seven steps tailored to develop module-sized programs of 50 to 5,000 LOC. Each step has a set of associated scripts, forms, and templates. During the course, engineers use the processes to complete the programming and report exercises shown in Table 1. As engineers learn to measure their work, analyze these measures, and set and meet improvement goals, they see the benefits of using defined methods and are motivated to consistently use them. The 10 PSP exercise programs are small: the first eight average 100 LOC and the last two average 200 and 300 LOC, respectively. Completing these programs, however, takes a good deal of work. While a knowledgeable instructor can substantially assist the students, the principal learning vehicle is the experience the students gain in doing the exercises.

When properly taught, the PSP

- ♦ demonstrates personal process principles,
- ♦ assists engineers in making accurate plans,
- ♦ determines the steps engineers can take to improve product quality,
- ♦ establishes benchmarks to measure personal process improvement, and
- ♦ determines the impact of process changes on an engineer's performance.

The PSP introduces process concepts in a series of steps. Each PSP step, shown in Figure 1, includes all the elements of prior steps together with one or two additions. Introducing these concepts one by one helps the engineers learn disciplined personal methods.

Personal Measurement (PSP0) is where the PSP starts. In this first step, engineers learn how to apply the PSP forms and scripts to their personal work. They do this by measuring development time and defects (both injected and removed). This lets engineers gather real, practical data and gives them benchmarks against which they measure progress while learning and practicing the PSP. PSP0 has three phases: planning, development (which includes design, code, compile, and test), and postmortem. PSP0.1 adds a coding standard, size measurement, and the Process Improvement Proposal form. The PIP form lets engineers record problems, issues, and ideas to use later in improving their processes. They also see how forms help them to gather and use process data.

Personal Planning (PSP1) introduces the PROBE method. Engineers use PROBE to estimate the sizes and development times for new programs based on their personal data.[2] PROBE uses linear regression to calculate estimating parameters, and it generates prediction intervals to indicate size and time estimate quality. Schedule and task planning are added in PSP1.1. By introducing planning early, the engineers gather enough data from the 10 PSP exercises to experience the benefits of the PSP statistical estimating and planning methods.

Personal Quality (PSP2) introduces defect management. With defect data from the PSP exercises, engineers construct and use checklists for design and code review. They learn why it's important to focus on quality from the start and how to efficiently review their programs. From their own data, they see how checklists can help them effectively review design and code as well as how to develop and modify these checklists as their personal skills and practices evolve. PSP2.1 introduces design specification and analysis techniques, along with defect prevention, process analyses, and process benchmarks. By measuring the time tasks take and the number of defects they inject and remove in each process phase, engineers learn to evaluate and improve their personal performance.

Scaling Up (PSP3) is the final PSP step. Figure 2 illustrates how engineers can couple multiple PSP2.1 processes in a cyclic fashion to scale up to developing modules with as many as several thousand LOC. At this PSP level, engineers also explore design-verification methods as well as process-definition principles and methods.

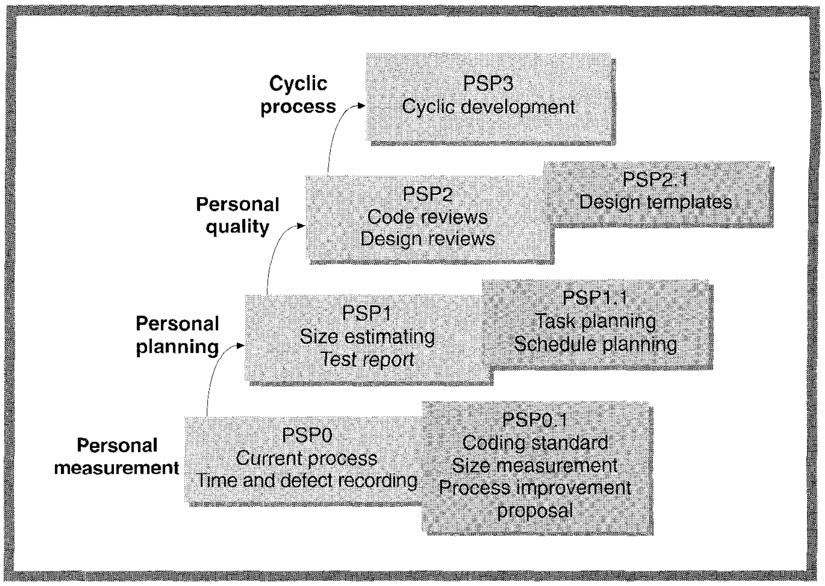


Figure 1. *PSP process evolution.*

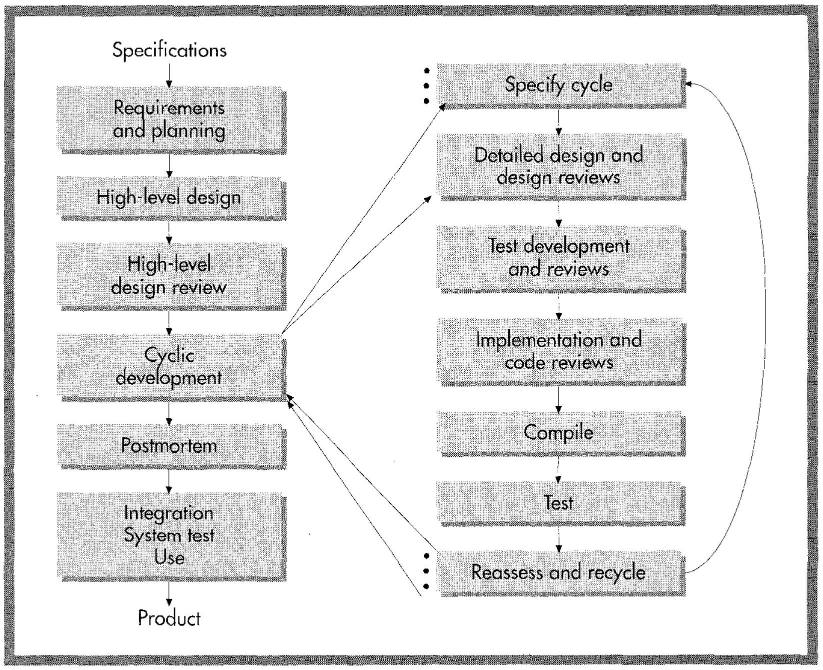


Figure 2. *PSP3 process.*

PSP/CMM relationship. The CMM is an organization-focused process-improvement framework.[1,3] While the CMM enables and facilitates good work, it does not guarantee it. The engineers must also use effective personal practices.

This is where the PSP comes in, with its bottom-up approach to process improvement. PSP demonstrates process improvement principles for *individual engineers* so they see how to efficiently produce quality products. To be fully effective, engineers need the support of a disciplined and efficient environment, which means that the PSP will

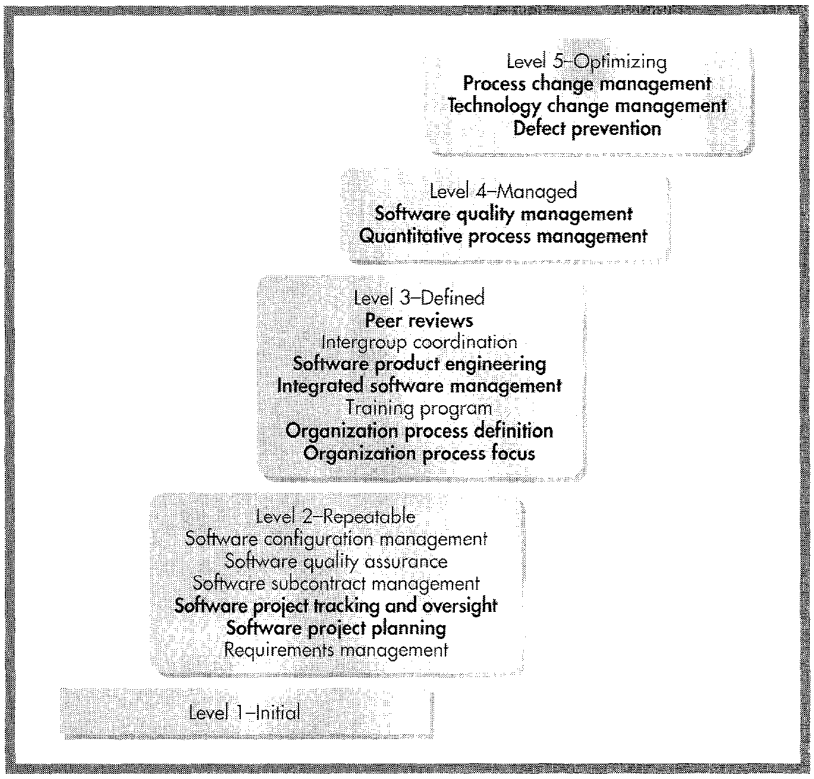

Level 5–Optimizing
Process change management
Technology change management
Defect prevention
Level 4–Managed
Software quality management
Quantitative process management
Level 3–Defined
Peer reviews
Intergroup coordination
Software product engineering
Integrated software management
Training program
Organization process definition
Organization process focus
Level 2–Repeatable
Software configuration management
Software quality assurance
Software subcontract management
Software project tracking and oversight
Software project planning
Requirements management
Level 1–Initial

Figure 3. PSP elements in the Capability Maturity Model, highlighted in bold face.

be most effective in software organizations near or above CMM Level 2.

The PSP and the CMM are mutually supportive. The CMM provides the orderly support environment engineers need to do superior work, and the PSP equips engineers to do high-quality work and participate in organizational process improvement. As Figure 3 shows, the PSP demonstrates the goals of 12 of the 18 CMM key process areas. The PSP demonstrates only those that can be accommodated with individual, classroom-sized exercises.

PSP development. In the initial PSP experiments, I wrote 61 Pascal and C++ programs using a personal process as near to meeting the goals of CMM Level 5 as I could devise. I also applied these same principles to personal financial work, technical writing, and process development. This work showed me that a defined and measured personal process could help me do better work and that programming development is a compelling vehicle for introducing personal process management. A more complex process than other personal activities, software development potentially includes many useful measures that can provide engineers an objective evaluation of their work and the quality of their products.

After the initial experiments, I needed to demonstrate that the PSP methods could be effectively applied by other software engineers, so I had two graduate students write several programs using an early PSP version. Because this early PSP was introduced all at once in one step, the students had difficulty. They tried some parts of the PSP and ignored others, which meant they did not understand the overall process and could not measure its effect on their personal performance. This experiment convinced me that process introduction was important; thus, the seven-step strategy evolved.

Early industrial PSP experiments corroborated the importance of process introduction. Various groups were willing to experiment with the PSP but until these methods were introduced in an orderly phased way no engineer consistently used the PSP. In one group, for example, project engineers defined personal and team processes and committed to use them. Although a few gathered some process data and tried several methods, no one consistently used the full process. The problem appeared to be the pressure the engineers felt to complete their projects. Management had told them that using the PSP was more important than meeting the project schedules, but they still felt pressured and were unwilling to use unfamiliar methods.

Learning new software methods involves trial and error, but when faced with deadlines engineers are reluctant to experiment. While they might intellectually agree that a new practice is an improvement, they are reluctant to take a chance and generally fall back on familiar practices.

I was thus faced with a catch-22. Without data, I couldn't convince engineers to use the PSP. And unless engineers used the PSP, I couldn't get data. Clearly, to obtain industrial experience, I needed to first convince engineers that the PSP methods would help them do better work, so I decided to introduce the PSP with a graduate university course. By introducing PSP methods one at a time and with one or two exercises for each, this course would give engineers the data to demonstrate how well the PSP worked, without the pressure of project schedules.

PSP METHODS

Among the software-engineering methods PSP introduces are data gathering, size and resource estimating, defect management, yield management, cost of quality, and productivity analysis. I discuss these methods here with some examples that merge data for several PSP classes and for multiple programming languages. As you can see from the statistical analysis box on page 81, it makes sense to pool the PSP data in this manner.

STATISTICAL ANALYSIS OF PSP DATA

The analysis of variance test was applied to data for 88 engineers from eight PSP classes. Program sizes, development times, and numbers of defects found were all separately tested. The results are shown in Table A. Since F(80, 7) at 5 percent is 3.29, the null hypothesis cannot be rejected in any of the program 1 cases. For the analyses in this article, the various class data are thus treated as a single set.

Data for program 10, the last PSP exercise, was similarly examined. Here, the population examined was 57 engineers from five courses. The smaller population was used because two of the eight courses only completed nine of the exercises and one course had not completed at the time of the analysis. As Table A shows, the analysis of variance test showed that the null hypothesis could not be rejected. In this case, F(52, 4) at 5 percent is 5.63. Again, for this article, data from all the PSP classes are thus pooled for the analyses.

The analysis of variance test also examined potential performance differences caused by six different programming languages used in the PSP classes to date. Only three had substantial use, however: C was used by 46 engineers, C++ by 21, and Ada by 8. The other languages were Fortran, Visual Basic, and Pascal. Only data on C, C++, and Ada were tested. As Table A shows, the variances among individuals were substantially greater than those among languages, so the null hypothesis cannot be rejected and the data for all languages are pooled. Here, F(72, 2) at 5 percent is 19.5.

The Wilcoxon matched-pairs signed-rank test examined the significance of the changes in the engineers' performance between programs 1 and 10. The comparison was made for one class of 14 engineers for total defects found per KLOC, defects per KLOC found in compiling, and defects per KLOC found in testing. The T values obtained in these cases were 5, 1, and 0 respectively. For N=13 and 0.005 significance in the one tailed test, T should be less than 9. In all cases, these improvements thus had a significance of better than 0.005. A repeated measures test has also been run on these same parameters and all the changes were found to be significant.

Table A
Analysis of Variance — F Values

	Ratio of Variances		
Measure	**Program 1**	**Program 10**	**Languages**
Program size	2.09	2.37	0.450
Development time	1.20	1.87	1.460
Number of defects	0.65	3.31	0.042

Gathering data. The PSP measures were defined with the Goal-Question-Metric paradigm.[4] These are the time the engineer spends in each process phase, the defects introduced and found in each phase, and the developed product sizes in LOC. These data, gathered in every process phase and summarized at project completion, provide the engineers a family of process quality measures:

- ♦ size and time estimating error,
- ♦ cost-performance index,
- ♦ defects injected and removed per hour,
- ♦ process yield,
- ♦ appraisal and failure cost of quality, and
- ♦ the appraisal to failure ratio.

Estimating and planning. PROBE is a proxy-based estimating method I developed for the PSP that lets engineers use their personal data to judge a new program's size and required development time. Size proxies, which in the PSP are objects and functions, help engineers visualize the probable size of new program components. Other proxies — function points, book chapters, screens, or reports — are also possible.

The PSP estimating strategy has engineers make detailed size and resource estimates. Although individual estimates generally have considerable error, the objective is to learn to make unbiased estimates. By coupling a defined estimating process with historical data, engineers make more consistent, unbiased estimates. When engineers estimate a new development in multiple parts, and when they make about as many overestimates as underestimates, their total project estimates are more accurate. The estimating measure is the percentage by which the final size or development time differs from the original estimates.

Overall, engineers' estimating ability improved moderately during the PSP course. At the beginning, only 30.8 percent of 104 engineers estimated within 20 percent of the correct program size. For program 10, 42.3 percent did. For time estimates, 32.7 percent of these 104 engineers estimated within 20 percent of the correct development time for program 1 and 49.0 percent did for program 10.

In general, individual estimating errors varied widely. Some engineers master estimating skill more quickly than others, so it was no surprise that some engineers improved considerably while others did not. Even though 10 exercises can help engineers understand estimating methods, they generally need more experience both to build an adequate personal estimating database and to gain estimating proficiency. These data suggest, however, that by using PROBE most engineers can improve their ability to estimate both program size and development time.

Planning accuracy is measured by the *cost-performance index*, the ratio of planned to actual development cost. For the PSP course, engineers track the cumulative value of their personal CPI through the last six exercises.

Managing defects. In the PSP, all defects are counted, including those found in compiling, testing, and desk checking. When engineers do inspections, the defects they find are also counted. The reason to count all defects

TABLE 2
PSP DEFECT TYPES

Type Number	Type Name	Description
10	Documentation	comments, messages
20	Syntax	spelling, punctuation, typos, instruction formats
30	Build, package	change management, library, version control
40	Assignment	declaration, duplicate names, scope, limits
50	Interface	procedure calls and references, I/O, user formats
60	Checking	error messages, inadequate checks
70	Data	structure, content
80	Function	logic, pointers, loops, recursion, computation, function defects
90	System	configuration, timing, memory
100	Environment	design, compile, test, or other support-system problems

is best understood by analogy with filter design in electrical engineering. If you examine only the noise output, you cannot obtain the information needed to design a better filter. Finding software defects is like filtering noise from electrical signals — the removal process must be designed to find each defect type. Logically, therefore, engineers should understand the defects they inject before they can adjust their processes to find them.

A key PSP tenet is that defect management is a software engineer's personal responsibility. If you introduce a defect, it is your responsibility to find and fix it. If the defects are not managed like this, they are more expensive to find and fix later on.[1]

As engineers learn to track and analyze defects in the PSP exercises, they gather data on the *phases* when the defects were injected and removed, the *defect types*, the *fix times*, and *defect descriptions*. Phases are planning, design, design review, code, code review, compile, and test. The defect types, shown in Table 2, are based on Ram Chillarege's work at IBM Research.[5] The fix time is the total time from initial defect detection until the defect is fixed and the fix verified.

Defect trends for 104 engineers are shown in Figure 4. These are the engineers in the PSP classes for whom I have data and who have completed the 10 programming exercises. (Other engineers met these criteria but reported either incomplete or obviously incorrect results.) Of the 104 engineers, 80 took the PSP in university courses and 24 in industrial courses. Of the 80 university students, 16 were working engineers taking a night course and 28 were working engineers earning a graduate degree by returning to school full-time. Thus more than half of the engineers in this sample had worked in software organizations.

The top line in Figure 4 shows the average of the total number of defects found for each of the exercises. With program 1, the average is 116.4 defects per KLOC with a standard deviation of 76.9. By program 10, the average number of defects had declined to 48.9, and the standard deviation narrowed to 35.5.

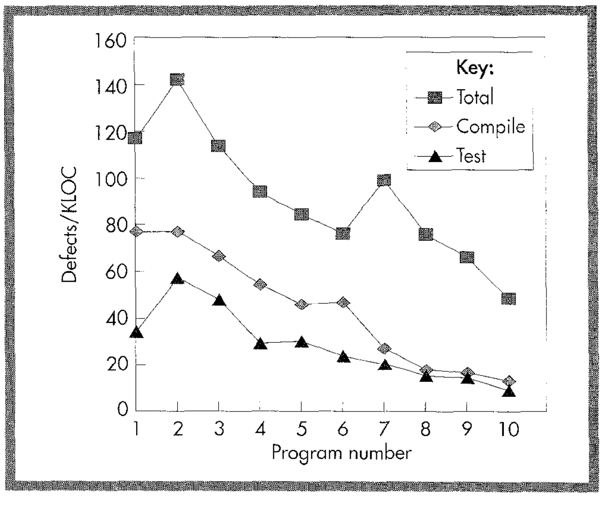


Figure 4. Defects per KLOC trend.

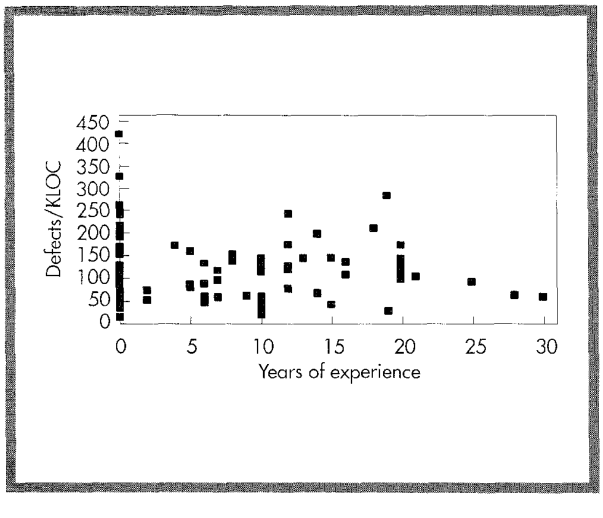


Figure 5. Defects versus experience, program 1.

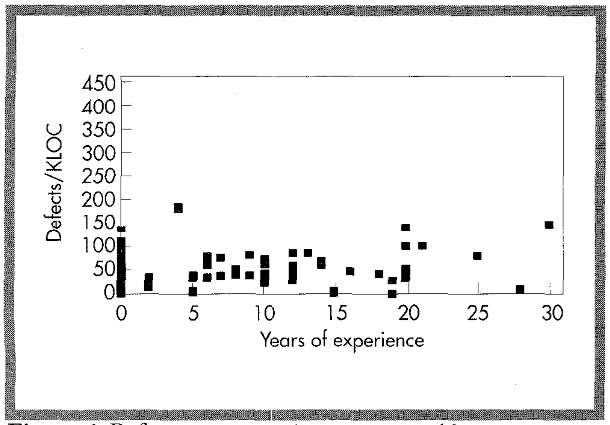


Figure 6. Defects versus experience, program 10.

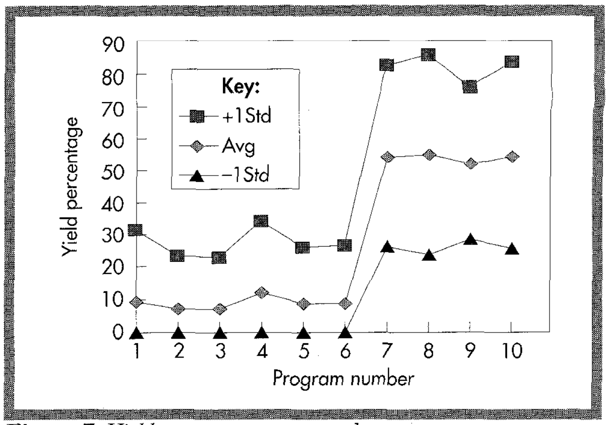


Figure 7. Yield versus program number.

The middle line in Figure 4 shows fewer defects found in compiling, from an average of 75.5 to 12.7 per KLOC, which is an improvement of about six times. The standard deviation narrowed from 58.7 to 12.7. For defects found in testing, the bottom line in Figure 4 shows reduced average defect levels, from 33.8 to 9.5 per KLOC, and reduced standard deviation, from 33.8 to 12.0.

Almost half (41) of these 104 engineers completed questionnaires, and the demographic data shows a modest relationship between defects per KLOC and years of engineering experience. While there is considerable variation in the defect rates for program 1, Figure 5 shows that the engineers with more than 20 years experience had somewhat lower defect rates than many less-experienced engineers, some of whom had low rates while many did not. As shown in Figure 6, the relationship between defect rates and experience does not hold for program 10. In fact, it appears that the less-experienced engineers learned the PSP methods better than their more experienced peers. Plots of defect levels versus both total LOC written and LOC written in the previous 12 months show no significant relationships.

Managing yield. Yield is the principal PSP quality measure. Total process yield is the percentage of defects found and fixed before the engineer starts to compile and test the program. Although software quality involves more than defects, the PSP focuses on defect detection and prevention because finding and fixing defects absorbs most of the development time and expense. When they start PSP training, engineers spend about 30 percent of their time compiling and testing programs, which probably mirrors their actual work practice. When engineers release actual modules for integration and system test, software organizations devote another 30 to 50 percent of development time in those phases,[6] almost exclusively to find and fix defects. Thus, despite other important quality issues, defect management will receive priority, at least until defect detection and repair costs are reduced.

If engineers want to find fewer defects in test, they must find them in code reviews. If they're going to review the code anyway, why not review it before compiling? This saves time they would have spent in compiling, and the compiler will act as a quality check on the code reviews. With few exceptions, however, engineers must first be convinced by their own data before they will do thorough design and code reviews prior to compiling.

In PSP, engineers must review their code before the first compile. Engineers often think the compiler's efficiency at finding syntax errors means they needn't bother finding them in reviews. However, some syntax defects will not be detected, not because the compilers are defective, but because some percentage of erroneous keystrokes will produce "valid" syntax that is not what the engineer intended. These defects cannot be found by the compiler and can be difficult to find in test. PSP data indicates that 9.4 percent of C++ syntax defects escape the compiler. If these defects are not found before compiling, they can take 10 or more times as long to find in unit test and, if not found in unit test, can take many hours to find in integration test, system test, or system operation.

The satisfaction that comes from doing a quality job is another reason to review code before compiling. Engineers like finding defects in code review, and they get great satisfaction from a clean first compile. Conversely, when they find few defects in code review, they feel they wasted their time. My personal experience also suggests that projects whose products have many defects in compile tend to have many defects in test. These projects also tend to be late and over budget.

Evidence shows that the more defects you find in compile, the more you are likely to find in test. Data on 844 PSP programs from 88 engineers show a correlation of 0.711 with a significance of better than 0.005 between the numbers of defects found in compile and those found in test. Thus, the fewer defects you find in compile, the fewer you are likely to have in test.

Reduced numbers of test defects imply a higher quality-shipped product. While it could be argued that finding few defects in test indicates poor testing, limited data show high correlation between the numbers of defects found in test and the numbers of defects later found by users. Martin Marietta, for example, has found a correlation of 0.911.[7]

Figure 7 shows the yield trends for our 104 engineers. Here, the sharp jump in yield with program 7 results from the introduction of design and code reviews at that point.

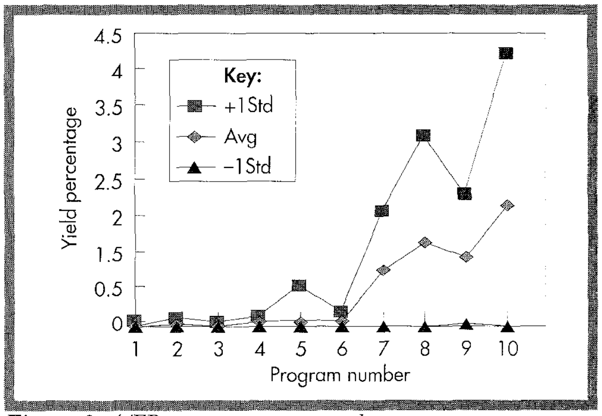


Figure 8. A/FR versus program number.

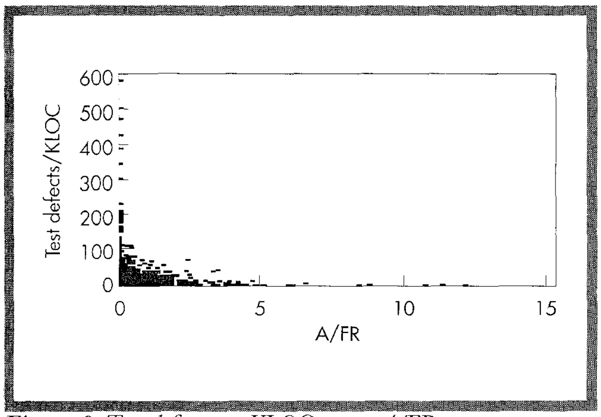


Figure 9. Test defects per KLOC versus A/FR.

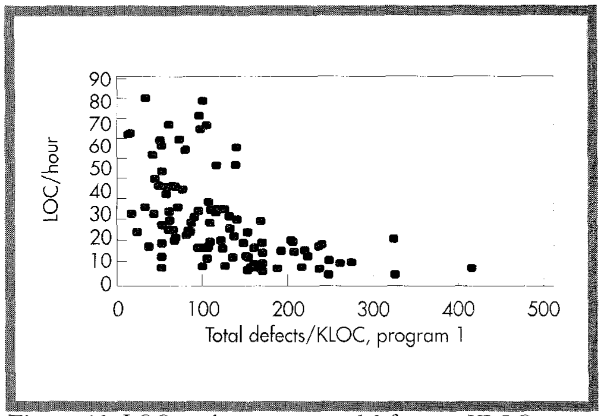


Figure 10. LOC per hour versus total defects per KLOC.

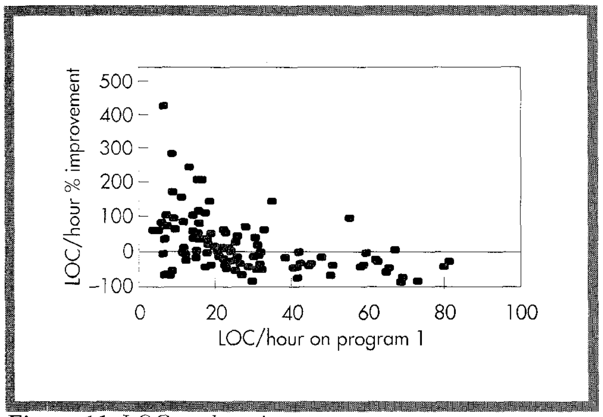


Figure 11. LOC per hour improvement.

Controlling cost of quality. To manage process quality, the PSP uses three cost-of-quality measures:

♦ appraisal costs: development time spent in design and code reviews,

♦ failure costs: time spent in compile and test, and

♦ prevention costs: time spent preventing defects before they occur. Prevention costs include prototyping and formal specification, methods not explicitly practiced with the PSP processes.

Another cost-of-quality measure is the ratio of the appraisal COQ to the failure COQ, known as the appraisal-to-failure-ratio. The A/FR is calculated by dividing the appraisal COQ by the failure COQ, or the ratio of design and code review time to compile and test time. The A/FR measures the relative effort spent in early defect removal. While the yield objective is to reduce the number of defects found in compile and test, the A/FR objective is to improve yield.

Figure 8 shows the improvement in A/FR for the same 104 engineers. Notably, A/FR increases with exercise 7 when design and code reviews are first introduced. Figure 9 shows data on A/FR and test defects for the 1,821 programs for which I have data. Here, A/FR values above 3 are associated with relatively few test defects while A/FRs below 2 are associated with relatively many test defects. PSP's suggested strategy is that engineers initially strive for A/FR values above 2. If they continue to find test defects, they should seek higher A/FR values. Once they consistently find few or no test defects, they should work to reduce A/FR while maintaining a high process yield.

Achieving higher product quality is the reason to increase A/FR. Once the quality objective is met, A/FR reductions will increase productivity. Since engineers generally cannot determine product quality during development, the A/FR measure is a useful guide to personal practice. By striving to increase their A/FR, engineers think more positively about review time. This helps them reduce compile and test time, and it reduces defects found in test.

The difference in time the engineers spend in compile and test shows how effective the A/FR measure can be. In one class, 75 percent of the engineers spent more than 20 percent of their time compiling and testing program 1. On program 10, only 8 percent did. Similarly, with program 1, no engineer spent less than 10 percent of the time compiling and testing, while with program 10, 67 percent did.

Understanding productivity. PSP-trained engineers learn to relate productivity and quality. They recognize that it makes no sense to compare the productivity of one programming process that found no test defects with one that had many. Defect-filled code will likely require many hours in integration and system test. Conversely, once engineers learn to produce defect-free (or nearly so) programs, their projects will likely be more productive.

Figure 10 shows the LOC/hour rate achieved with PSP program 1 by the 104 engineers. From these data, higher defect content appears associated with lower LOC/hour rates (productivity). Note, however, that low defect content by itself did not guarantee high productivity. This relationship is even more pronounced with program 10: Those engineers who injected the most defects had the lowest LOC/hour development rates.

Figure 11 shows the improvement in LOC/hour for the group of 104 engineers. Engineers with the highest LOC/hour rates on program 1 usually had no improvement. Although the productivity of this group improved by an average of 20.84 percent, it is clear that many engineers who had high LOC/hour rates for program 1 had lower rates for program 10. This suggests two conclusions:

♦ Because many inexperienced engineers initially have higher defect rates and lower LOC/hour rates, the PSP disciplines will most likely increase their LOC/hour rates. They will then see the PSP as helping them to work faster and will probably continue using PSP methods.

♦ Some experienced engineers start with low defect rates and high LOC/hour rates. When these engineers add the PSP estimating and planning tasks, follow defined coding standards, review their programs, and track and report their results, their LOC/hour rates will often drop. These engineers will then see the PSP as slowing them down; if they do not appreciate the benefits of these planning and quality-management practices, they will probably stop using the PSP.

Engineers do not normally do several major tasks featured in the PSP, so it is not surprising that, when these tasks are added, some engineers end up with lower LOC/hour rates. Writing module-sized programs is a little like running a four-minute mile. When engineers can produce 40-plus LOC per hour, where will improvement come from? This focus on maximizing engineers' personal rates, however, leads to suboptimization. The PSP management and planning methods take time, but these are the very methods that make software engineers effective organizational team members. By taking the time to follow disciplined personal methods, they produce higher quality programs. When their programs have fewer defects, they require less time in integration and system test. The engineers' more disciplined work thus prepares them to develop high-quality large programs.

OTHER PSP ISSUES

Software design, process scale-up, and process definition are also addressed in the PSP.

Design. PSP's principal design focus is preventing design defects. The PSP approach is to use design-completion criteria, rather than advocating specific design methods. PSP research shows that defects result mainly from oversights, misunderstandings, and simple goofs, not complicated logic designs. Many defects are caused by improperly represented designs, incomplete designs, or no design at all. Moreover, poor design representation can cause engineers to design during implementation, which can be a significant source of error. By establishing design completion criteria, therefore, the PSP helps engineers produce reviewable designs that can be implemented with minimum error.

PSP data also show that engineers inject about 3.5 times as many defects

RESOURCES FOR EDUCATORS AND TRAINERS

The following materials are available from Addison-Wesley Publishing Company, Reading, Mass:

♦ Instructor's Guide for *A Discipline for Software Engineering*. This is free to people who teach or plan to teach a PSP course with this textbook. It contains the course outline, lecture suggestions, data presentation guidelines, grading criteria, instructions for the instructors' spreadsheets, and copies of the homework assignment kits.

♦ Instructor's Diskette for *A Discipline for Software Engineering*. This diskette is free to PSP instructors. It contains 701 lecture overheads for the 15 course lectures, spreadsheets for analyzing and graphing student data, and spreadsheet instructions.

♦ Support Diskette for *A Discipline for Software Engineering*. This contains a spreadsheet for each student to use to enter and graph exercise data, a summary spreadsheet to simplify exercise reporting, and the assignment kits.

Additional material and information can be obtained electronically from Addison-Wesley via Internet at gopher aw.com or via the World Wide Web at http://www.aw.com/cseng/. Look under book and author.

The SEI offers industrial PSP courses:

♦ PSP Instructor Training
♦ Intro to PSP
♦ Advanced PSP
♦ PSP for Managers

The SEI provides additional information on PSP publications and industrial courses on the World Wide Web at SEI http://www.sei.cmu.edu.

per hour during coding as they do during design. When engineers can save implementation time by producing better designs, they inject fewer defects and increase their productivity.

Although the PSP does not define generalized design completion criteria, it does offer an approach through four design templates that help engineers determine when their design is complete. The template structure is based on Dennis de Champeaux's[8] proposed object definition framework:

♦ *Internal-static.* Contains a static picture of the object, such as its logical design. For this, the PSP provides a logic-specification template.

♦ *Internal-dynamic.* The object's dynamic characteristics concerning its behavior. The dynamic behavior of an object can be described by treating it as a state machine. For this, the PSP provides a state-specification template. Other possibly important dynamic characteristics are response times and interrupt behavior.

A principle PSP objective is to extend to larger programs the productivity experienced in small programs.

♦ *External-static.* The static relationship of this object to other objects. For this, the PSP provides a function-specification template, which includes the inheritance class structure.

♦ *External-dynamic.* The interactions of this object with other entities. An example would be the call-return behavior of each of the object's methods. For this, the PSP provides an operational scenario template.[2]

Scale-up. The PSP's objective is to extend to larger programs the productivity engineers typically experience with small program development. The final PSP step, PSP3, follows the spiral-like process shown in Figure 2. After subdividing the large program into smaller elements, each element is developed with a PSP2.1-like process. These elements are then progressively integrated into the completed product.

Process definition. In helping engineers learn to apply process principles, the PSP shows them how to define new processes, how to plan a process-definition task, and how long such work typically takes. In the middle of the course, engineers are assigned the task of defining a process for analyzing process data and writing a report on their findings. They enact this process and submit the report they produce, their process definition, and work data.

At course end, the engineers update the midterm process to fix previously encountered problems and extend the process to include the more sophisticated analyses required for a second report. From these exercises, they see that process definition is straightforward and applicable to many tasks besides program development, including requirements definition, system test, program enhancement, and documentation development.

TRANSITIONING THE PSP INTO PRACTICE

Our focus now at the SEI is on transitioning the PSP into general practice through academic and industrial introduction.

Academic introduction. The initial PSP course was aimed at first-year graduate software engineers largely because I believed the students would have the required programming language proficiency and software development competence. The one-semester course is designed for 15 90-minute lectures. The standard PSP course assigns the 10 A-series programs listed in Table 1; the B series is optional. Because the full A-series course takes about 150 to 200 hours of an engineer's time, the 15-week class schedule represents a heavy workload. The time could be extended, depending on the academic schedule and whether or not other materials are introduced. It is essential, however, that engineers understand at the outset the amount of work involved.

The primary learning mechanism is the engineer's experience in completing the exercises. Frequent discussion of overall class data is necessary, but no individual engineer's data are shown to anyone except that engineer.

We are also experimenting with the PSP concepts in the undergraduate software-engineering curriculum. If the PSP were taught during their earliest courses, engineers would have the maximum opportunity to practice and perfect these methods before they started professional work. Based on the PSP experience, inexperienced engineers are more likely to find that the PSP discipline improves their performance, and they are then more likely to continue using these methods.

College juniors and seniors have completed the current PSP course with apparent success. PSP concepts are within the intellectual grasp of most college freshmen, however, so an undergraduate course textbook (now in test) is being prepared.[9]

Industrial introduction. Introducing the PSP into industrial organizations appears to be most successful in a course format. Individual self-study has been tried, but only about one in five to

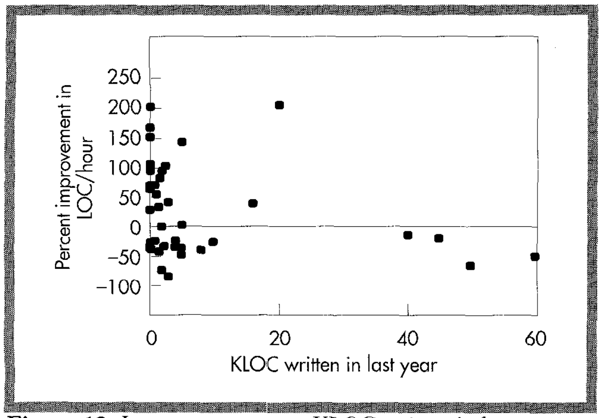


Figure 12. Improvement versus KLOC written in last year.

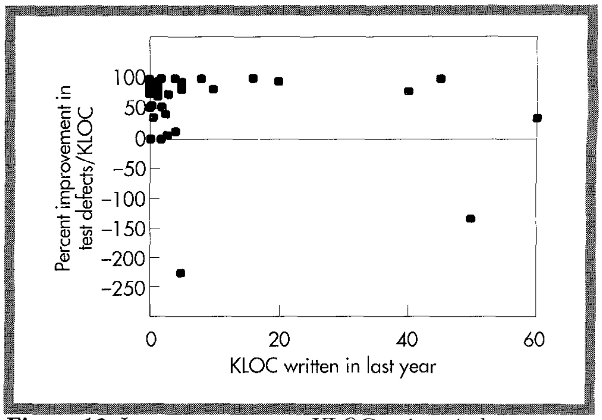


Figure 13. Improvement versus KLOC written in last year.

10 of the engineers who start such a course actually completes it. One industrial group of three engineers and a manager has taken the course as a team, with success. At latest report, this group is now starting to use the PSP on their project.

Another successful approach is to introduce the PSP from the top down in a course taught to the top management team, then to the engineers who work for them. In the one case that has been tried, two laboratory technical directors and the laboratory management team took the first course. Courses are next being given to a class that includes project engineers and leaders. Because the managers understood the work involved, they could convince their teams to take the PSP course. With their PSP background, the managers also appreciated the methods their people would be using and will be better equipped to lead their teams after PSP training.

Currently, the SEI offers several types of PSP training (see the box on p. 85). The SEI is also working with several corporations to determine the PSP's impact on organizational performance and is gathering data on engineers' backgrounds, the tools and methods used, and organizational performance. In addition, various techniques are being explored to determine how PSP affects different organizational quality and productivity indicators. It will likely take several years to complete these studies.

LANGUAGE FLUENCY AND IMPROVEMENT

Is some of the PSP improvement due to the programming fluency the engineers gain while completing the programming exercises? To find out, the SEI developed a questionnaire that asked the engineers to estimate how many LOC they had developed in the preceding 12 months. Figure 12 compares the change in LOC/hour versus the LOC the engineers claimed to have written in the past year. Although it is unlikely that many engineers knew precisely the number of LOC they had written in the last year, it is likely that those who had written little or no code would give low numbers. From Figure 12, it does not appear that recent programming experience is a major factor in the PSP learning process.

Another successful approach is to introduce PSP from the top down.

Similarly, Figure 13 shows the improvement in test defects versus the LOC written in the last year. Again, the relationship appears insignificant and also suggests that the large improvement in PSP student performance cannot be explained by the increased language fluency gained by completing the exercises.

When more questionnaire data are gathered we will refine our statistical analyses of these questions.

PSP data show that engineers can substantially improve their performance by using a defined and measured personal process. By defining their tasks, measuring their work, and striving to produce the highest quality products, engineers find that their work is more predictable and their products have fewer defects. Results from the PSP work done to date show the following:

- The PSP is effectively taught in a university graduate course. With adequate management support, this same course format works in industry. In all cases, the key to learning the material is that the engineers do the course exercises and periodically analyze their exercise data. The PSP is a self-learning experience that provides engineers an appreciation of data gathering and process management.
- The improvement in average defect levels for engineers who complete the PSP course is 58.0 percent for total defects per KLOC and 71.9 percent for defects per KLOC found in test.
- With extensive PSP data supporting their estimates, engineers can better justify their plans and explain to their managers the logic behind their cost and schedule estimates. This in turn helps them make realistic commitments to, and negotiate them with, their management.

The PSP is a promising discipline, but many questions remain to be studied. Early indications are that improved PSP performance will result in improved engineering practices. This has not yet been demonstrated in industrial practice however and will be the next challenge. ◆

ACKNOWLEDGMENTS

I thank Howie Dow for his early work in teaching and demonstrating the PSP with his first class. Nazism Madjahavi's early support and encouragement helped me formulate the PSP and structure the course and textbook. Iraj Hirmanpour and the faculty of the Aviation Computer Science Department at Embry-Riddle Aeronautical University have made the PSP the required first course for all graduate software-engineering students, and they are assisting me in developing and testing the PSP course for freshmen. Girish Seshagiri and Pat Ferguson at the AIS Corporation have given me helpful feedback from their early experiences in introducing and using the PSP.

I also thank those who reviewed this article: Alan Christie, Suzanne Couturiaux, Howie Dow, Andy Huber, John Leary, Julia Mullaney, Jim Over, Bob Park, Lauren Quinn, Erik Riedel, Neal Reizer, Glenn Rosander, Dan Roy, Jim Rozum, Larry Votta, and Dave Zubrow. I also appreciate the help and advice that Will Hayes and Jim Herbsleb gave me with the statistical issues.

This work is supported by the US Department of Defense.

REFERENCES

1. W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.
2. W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, Mass., 1995.
3. M.C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading Mass., 1995.
4. V. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Engineering*, Nov. 1984, pp. 728-738.
5. R. Chillarege et al., "Orthogonal Defect Classification — A Concept for In-Process Measurements," *IEEE Trans. Software Eng.*, Nov. 1992, pp. 943-956.
6. B.W. Boehm, *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
7. J. Henry et al., "Improving Software Maintenance at Martin Marietta," *IEEE Software*, July 1994, pp. 67-75.
8. D. de Champeaux, D. Lea, and P. Faure, *Object-Oriented System Development*, Addison-Wesley, Reading, Mass., 1993.
9. W.S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, Reading, Mass., to appear.

Watts S. Humphrey founded the Software Process Program of the Software Engineering Institute and is currently an SEI fellow. The process program provides leadership in establishing advanced software-engineering processes, metrics, methods, and quality programs for the US government. Humphrey worked at IBM for more than 25 years, where his assignments included responsibility for commercial software development, managing the Endicott, N.Y., development laboratory, and directing IBM's programming quality and process work. In 1991, he served on the Board of Examiners for the Malcolm Baldrige National Quality Award. He has been issued five US patents and has authored four books.

Humphrey received a BS in physics from the University of Chicago, an MS in physics from the Illinois Institute of Technology, and an MBA from the University of Chicago. He is an IEEE fellow and a member of the ACM.

Address questions about this article to Humphrey at SEI, Carnegie Mellon University, Pittsburgh, PA 15213-3890; watts@sei.cmu.edu.