Software Engineering Institute | Carnegie Mellon

Why Don't They Practice What We Preach?

Watts S. Humphrey

One of the most intractable problems in software is getting engineers to consistently use effective methods. The Software Engineering Institute (SEI) has worked on this problem for a number of years and has developed effective methods for addressing it. This paper describes these methods and shows what they have accomplished with several hundred students and working engineers. After first describing the problem of changing engineers' practices, the paper discusses the logic engineers typically follow in deciding what methods to use. Next is a description of the Personal Software Process[SM] (PSP[SM]) and the Team Software Process[SM](TSP[SM]). Finally, the implications of SEI's experiences with the PSP and TSP are described, with particular emphasis on software engineering and computer science education.

Introduction

One of the most intractable problems in software is getting engineers to use effective methods. The list is endless: good design, design verification, code inspection, comprehensive unit test, and so on. Even when there is clear evidence that the methods help and even with strong pressure to use them, many engineers still do not do so. This is true for students and for practicing engineers and it seems to be true almost regardless of the engineers' experience and training.

One of the best examples of this is a graduate course I taught to 14 software engineering students. They had all had courses in formal methods and structured design but not one used these methods to write any of the 10 course exercise programs. When I pointed out that they all had made design errors and asked why they had not used the design methods they had learned, they said these programs were too small. While they may not have known how to use the methods for small programs, engineers invariably write large programs as collections of many small ones. Thus, if they do not use the best methods for writing small programs, they are unlikely to use them when writing large programs.

Based on our experience at the Software Engineering Institute (SEI), there are techniques for getting engineers to use more effective methods, but they are not simple and they take a substantial amount of work. This paper describes these techniques and shows what they have accomplished with several hundred students and working engineers. I first describe the problem of changing engineers' practices and then discuss the logic engineers typically follow in deciding what methods to use. I follow this with a description of two process frameworks the SEI has developed to convince engineers to use better methods. Finally, I describe the implications of our experiences for software engineering and computer science education.

INDUSTRIAL SOFTWARE PRACTICES

The general practices of industrial software engineers are poor by almost any measure. Their projects are typically late and over costs, they cannot predict when they will finish, and the final products frequently have many defects. While there are exceptions and software engineers are generally dedicated and hard working, this situation has existed for many years. It is therefore unlikely to significantly improve without major changes in the way we educate and manage software engineers and computer scientists. Briefly, the state of software practice has been so bad for so long because

- The educational system does not provide graduates with the practical skills they will later need. In addition to technical knowledge, they need to make accurate plans, to measure and manage the quality of their work, and to function effectively as development team members.
- Few software organizations are willing or able to provide the remedial training their new engineers need.

- Today's software organizations have few if any role models who consistently demonstrate effective work habits and disciplines.

Addressing the problem of poor working practices must largely be the responsibility of management in software organizations. The academic community, however, could be a major help if they were to focus more on how engineers work and somewhat less on the products they produce. Today, since students are not taught effective planning and quality management, they must learn these practices on their own or by emulating their peers. This perpetuates the common student ethic of ignoring planning, design, and quality in a mad rush to start coding. This craft-like approach for advancing the state of software practice has not been effective in the past and it will almost certainly not be effective in the future.

THE PROBLEM OF IMPROVING PERSONAL PRACTICES

Perhaps the most critical issue in improving the state of software practice is getting software engineers to use improved methods. This is a nontrivial problem, particularly because even intelligent people often will not do things that common logic, experience, and even hard evidence suggests that they should. Many software engineers thus do not use known best methods, even when their professors or managers tell them to do so.

The reasons for this both explain why process improvement is so difficult and suggest a logic for addressing the problem. In summary these reasons are:

1. Once engineers have learned how to develop small programs that work, they have also established some basic personal practices.
2. The more engineers use and reinforce these initial methods, the more ingrained they become and the harder they are to change.
3. Engineers have found that many impressive-sounding tools and techniques do not provide their promised benefits. It is therefore hard to convince them that some new method will help them to do better work.
4. Since no one generally observes the methods software engineers use, no one will know how they work. Thus engineers don't have to change their working methods if they don't want to.

The problems of improving the personal practices of software engineers had long interested me, so, when I had the opportunity, I started a study of how process improvement principles would apply to individual software work. I spent several years writing programs using as close to a Capability Maturity Model^SM (CMM®) Level 5 personal process as I could devise [Humphrey 1989, Paulk 1995]. The results were truly amazing. I was more productive, the quality of my work improved sharply, and I could make accurate personal plans. This process framework is what I call the Personal Software Process (PSP)^SM [Humphrey 1995].

The next issue was to see how effective these PSP methods would be for others. My first approach was to meet with several engineering groups to describe what I had done and get them to try it. This was a dismal failure. One laboratory manager even told his people that it was more important for them to use these methods than to meet their project schedules. Even though the engineers said they would do so, they still did not. The question was why not?

A QUESTION OF CONVICTION

When students first learn to write programs, they generally adopt the practices used by their peers and seniors. While the faculty may talk convincingly about the importance of thorough designs and orderly plans, their grades only depend on the programs they produce, not on how they produced them. As opposed to laboratory courses in chemistry, physics, or biology or the rigorous training of surgeons, software practices are largely ad-hoc and informal. Students thus become convinced that coding and testing are essential and everything else is optional. As they gain experience, it grows increasingly difficult to change their minds.

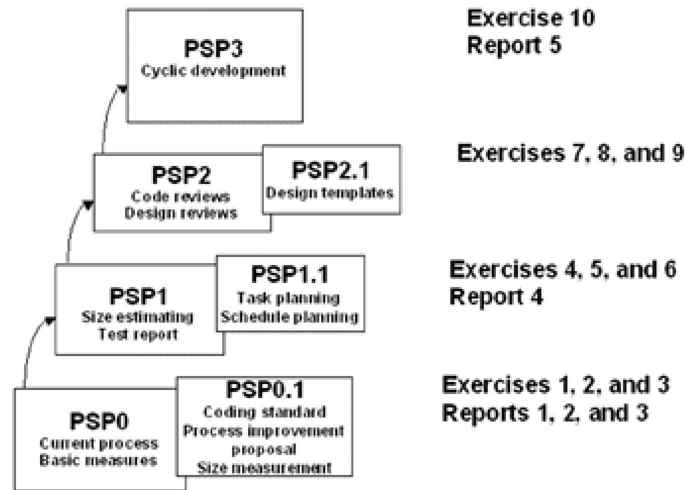
To change software engineers' personal practices, we must convince them that new methods are more effective than the way they currently work. This, however, is difficult because engineers will only believe that a new method works for them by using it and seeing the results. Convincing them to try new methods, however, is nearly impossible since they will not even try a new method until they believe it will work.

The Personal Software Process (PSP)

Given all this, how could we possibly convince engineers that a new method would work for them? The only way we could think of to change this behavior was with a major intervention. We had to actually expose the engineers to new ways of working. We thus decided to remove them from their day-to-day environment and put them through a rigorous training course. Here, as shown in Figure 1,

the engineers follow prescribed methods and write a defined set of programming exercises [Humphrey 1995]. With these exercises, we gradually introduce various advanced software engineering methods. By measuring their own performance, the engineers can see the effect of these methods on their work.

Figure 1. The PSP Process Evolution



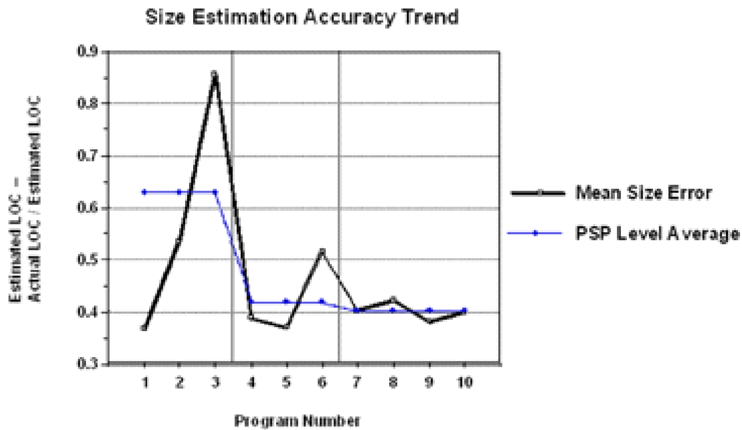
The PSP course works because it addresses the principal reasons students and engineers do not use improved methods:

1. Because they are in a course and their work is graded on their use of PSP methods, they must use these methods to satisfactorily complete the course.
2. From their personal data, they see how many defects they inject, how much time they waste in compiling and testing, and how much room they have for personal improvement.
3. As they use the PSP planning, quality, and design methods, they see from their personal experience that their work is both more predictable and more fun.

THE BENEFITS STUDENTS SEE

Figures 2, 3, 4, and 5 show some of the benefits the engineers experience [Hayes 1997, Humphrey 1996]. Figure 2 shows the average reduction in size estimating error for nearly 300 engineers who have taken the PSP course and provided data to the SEI. Their average size estimating error at the beginning of the course is shown at the left, and their error at the end of the course is shown at the right. This shows that size-estimating errors averaged 63% with PSP0 (the first three programs) and 40% with PSP2 and PSP3 (the last four exercises). Note that the PSP size estimating method (PROBE) is not introduced until PSP1, with program 4.

Figure 2. Size Estimation Results



For time estimating, Figure 3 shows an improvement from a 55% error to a 27% error, or a factor of about two. The improvement in compile and test defects, as shown in Figure 4, is the most dramatic change. From the first to the last of the 10 PSP programming exercises, the engineers' reduced their compile and test defects from 110 defects/KLOC on the first program to 20 defects/KLOC with program 10, an improvement of over five times. As shown in Figure 5, even with their greatly improved planning and quality performance, the engineers' productivity ended about where it started.

Figure 3. Effort Estimation Results

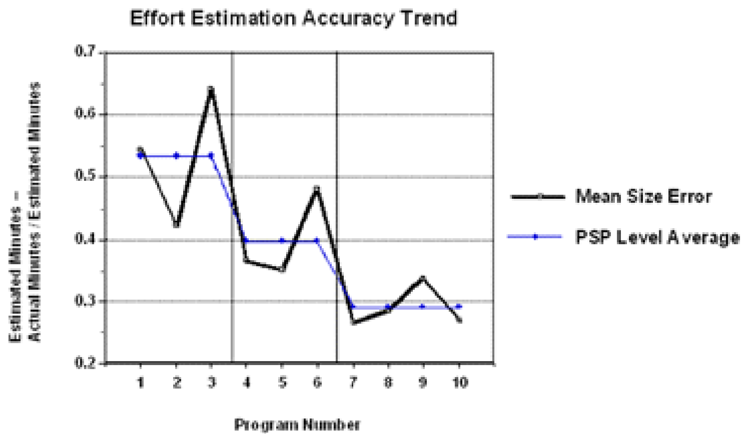


Figure 4. Quality Results

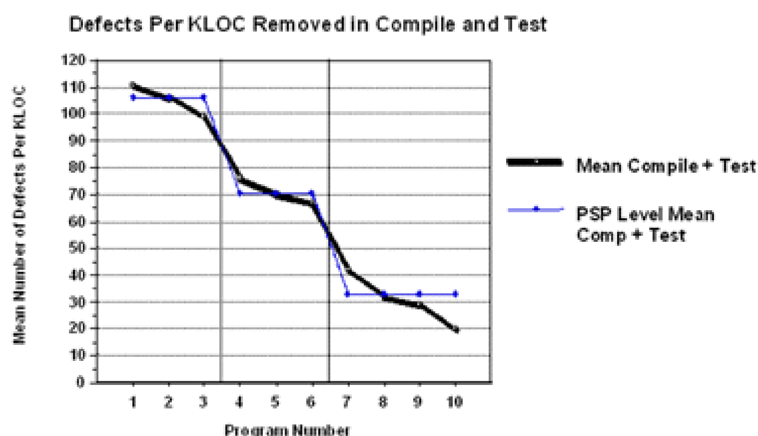
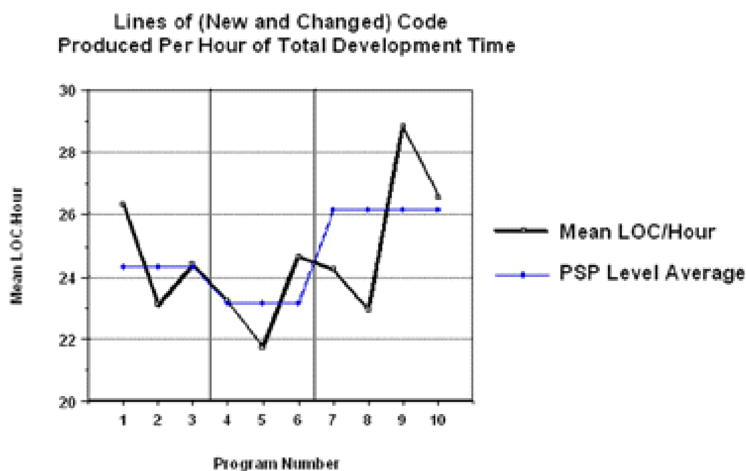
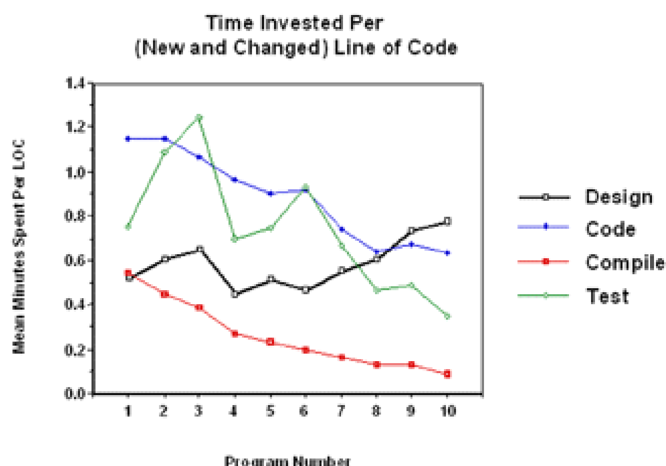


Figure 5. Productivity Results



From an education perspective, the most significant change is in the way the engineers spent their time. With program 1, as shown in Figure 6, this group of nearly 300 engineers spent on average less time designing their programs than they did on any other task. They even spent more time compiling than designing! At the end of the course, they spent more time designing than in any other activity. This is what we have been trying to get software engineers to do for years. Once they experience the benefits of doing thorough designs, their priorities change and they no longer concentrate on coding, compiling, and testing.

Figure 6. Effort Distribution Results



A final and perhaps most important point concerns improvement trends. From Figures 3, 4, 5, and 6, the rate of improvement generally continues right through the end of the course. It thus appears that the students could continue to improve if they had further practice. It is therefore important to introduce the PSP early in a computer science or software engineering curriculum and to require students to use it in all their subsequent courses.

INDUSTRIAL RESULTS WITH THE PSP

The PSP has now been used by a number of industrial organizations [Ferguson 1997]. From their data, the benefits of PSP training are evident. Figure 7 shows data from a team at Advanced Information Services (AIS), in Peoria, IL. They were PSP trained in the middle of their project. The three bars on the left show the engineers' time estimates for the weeks they expected to take to develop the first three components. For component 1, for example, the original estimate was 4 weeks and the job took 20. Their average estimating error for the first three components was 394%. These same engineers then took the PSP course and completed the remaining 6 components. On the right in Figure 7, their average estimating error after PSP training was -10.6%. The original estimate for component 8, for example, was 14 weeks and the work was completed in 14 weeks.

Figure 7. Schedule Estimating Error

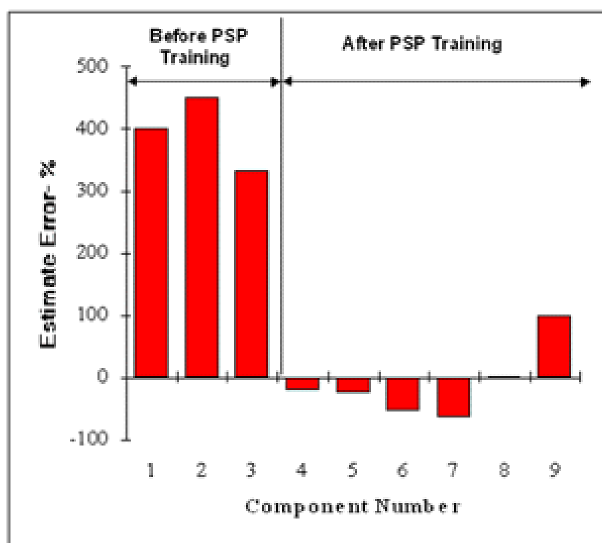


Table 1 shows acceptance test data on products from one group of AIS engineers. Before PSP training, they had a substantial number of acceptance test defects and their products were uniformly late. After PSP training, the very next product was nearly on schedule and it had only one acceptance test defect.

Table 2 shows the savings in system testing time for nine PSP projects. At the top, system test time is shown for several products that were completed before PSP training. At the bottom, system test time is shown for products that the same AIS engineers completed after PSP training. Note that A1 and A2 are two parts of the same product, so testing for them was done together in 1.5 months.

LESSONS FROM TEACHING THE PSP

Without continued management interest and support, we have found that few PSP-trained engineers will continue using these PSP methods. Without faculty emphasis, students also behave the same way. If faculty continue to grade students on their use of the PSP methods, the students will continue to use them. If their courses do not require these methods, few students will use them. There are, of course, always exceptions. Some engineers will never practice disciplined methods and others will continue to practice them even under adverse conditions. The majority of students and engineers, however, will follow what is in their pragmatic short-term interest. In software, unfortunately, the only generally accepted short-term priority is coding and testing.

Perhaps the most difficult change is for computer science professors. They need to shift their focus from the programs the students write to the data on the processes they use. Until faculty do this, they cannot properly evaluate the students' methods or guide them on improving their personal practices. Without this feedback, students will soon treat a PSP course like any other programming class. By carefully following the instructor's guides that accompany the PSP textbooks, faculty can avoid this problem [Humphrey 1995, Humphrey 1997]. The SEI and the Computer Science Department at Embry Riddle Aeronautical University (ERAU) also offer a one-week summer faculty workshop on teaching PSP courses.[1]

[1]Thomas B. Hilburn and James Over, Lectures and Exercises, PSP Faculty Workshop, Software Engineering Institute and Embry-Riddle Aeronautical University, Daytona Beach Florida, June 23-27, 1997.

The Team Software Process (TSP)

Because the successful introduction and use of the PSP depends greatly on the engineers working environment, we decided to look at the way software work is done by industrial teams. The approach we took was to incorporate the PSP methods into a larger process designed to guide software development teams. We subsequently broadened the objective to include combined hardware and software teams.

Software development is generally taught as a solo activity but when engineers go into industry, most of them work on teams. The transition from solo to team behavior is not obvious and engineers rarely get guidance on how to work as team members. The Team Software Process (TSP)[SM] provides such guidance. The TSP's principal objective is to show PSP-trained engineers how to run a team-based project. The TSP also creates an environment that fosters continued use of disciplined PSP methods. The TSP has five objectives:

1. Build self-directed teams who plan and track their work, establish goals, and own their own processes and plans. These can be pure software teams or integrated product teams (IPTs) of 2 to 20 PSP-trained engineers.
2. Show managers how to coach and motivate their teams and how to help them sustain peak performance.
3. Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
4. Provide improvement guidance to high-maturity organizations.
5. Facilitate university teaching of industrial-grade team skills.

TEAMBUILDING IS NOT OBVIOUS

Generally, when a group of engineers starts a project, they get little or no guidance on how to proceed. If they are lucky, their manager or one or two of the experienced engineers will have worked on well

run teams and have some idea of how to proceed. In most cases, however, the teams have to muddle through a host of issues on their own. Examples of some questions that every software team must address are:

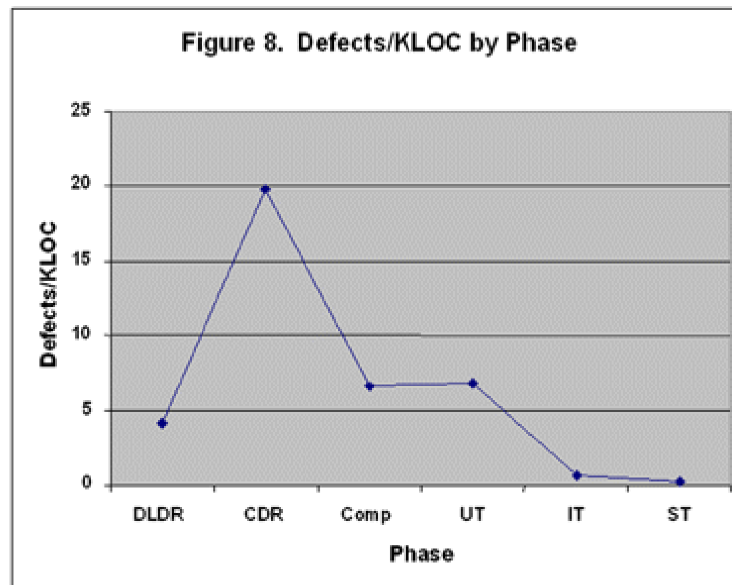
1. What are our goals?
2. What are the team roles and who will fill them?
3. What are the responsibilities of these roles?
4. How will the team make decisions and settle issues?
5. What standards and procedures does the team need and how do we establish them?
6. What are our quality objectives?
7. How will we track quality performance and what should we do if it falls short?
8. What processes should we use to develop the product?
9. What should be our development strategy?
10. How should we produce the design?
11. How should we integrate and test the product?
12. How do we produce our development plan?
13. How can we minimize the development schedule?
14. How do we assess, track, and manage project risks?
15. How do we determine project status?
16. How do we report status to management and the customer?

Most teams waste a great deal of time and creative energy struggling with these questions. This is unfortunate since none of these questions is new and there are known and proven answers for every one.

EARLY TSP EXPERIENCE

The TSP provides teams with explicit guidance on how to accomplish their objectives. All told, the TSP scripts lead engineers through 173 defined steps that show them what to do and how to do it.

The SEI is working with a few universities and a number of industrial organizations on introducing the TSP and the early results are encouraging. One ERAU student team removed over 99% of development defects before system test entry. Their defect-removal profile is shown in Figure 8. TSP teams also gather the data they need for a defect-prone analysis.



Figures 9 and 10 show component quality profiles for this same ERAU team. Component 7 had no integration or system test defects and component 9 had one integration defect. In the first several months of use, no defects have been found in this student-produced program.

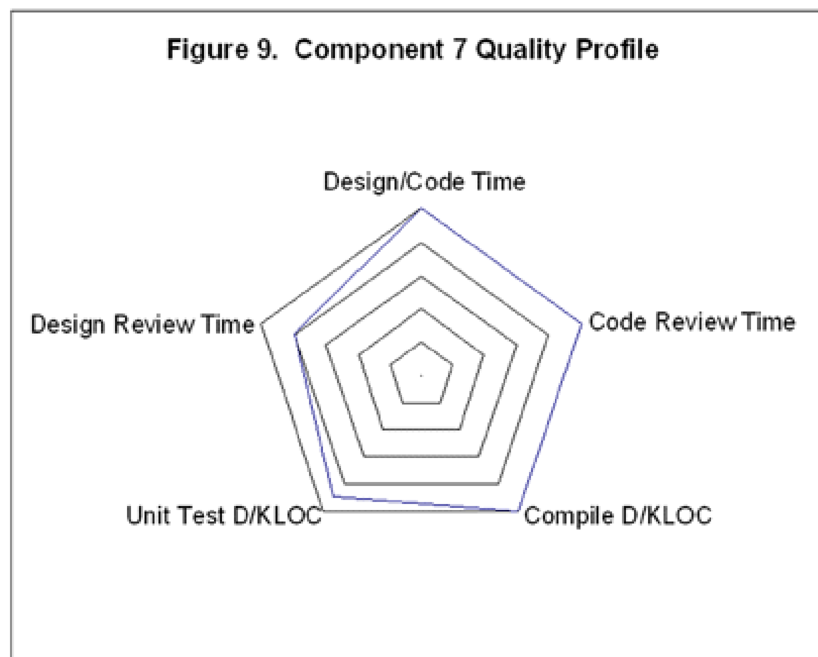


Figure 9. Component 7 Quality Profile

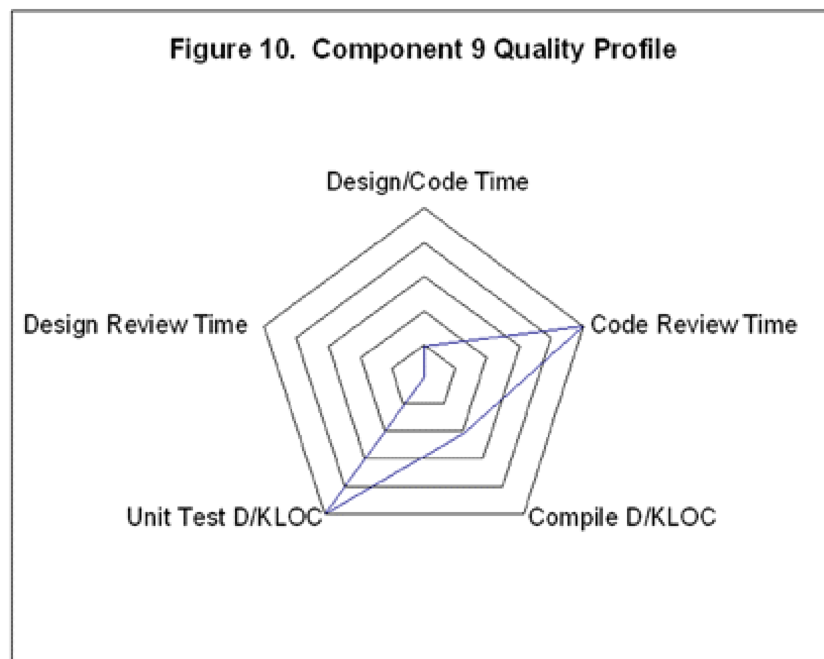


Figure 10. Component 9 Quality Profile

The quality profile provides a graphic way to identify components that have a high risk of having defects found in later testing or program use. The five profile dimensions are explained in Table 3. The various numerical criteria are based on PSP and TSP data. Each organization should adjust these criteria based on their own data and experience.

The quality profiles shown in Figures 9 and 10 plot each of these five profile dimensions on a radius. When the risk of defects is low, say, when there are less than 5 defects/KLOC in unit test, the unit test defects/KLOC profile is at 1, or the outer rim of the bullseye. When the risk is higher, say, at 10 unit test defects/KLOC, the risk is twice as high, as shown by a risk half way in towards the center of the bullseye. A much larger number of unit test defects would then be a profile value near the center.

Engineers can occasionally hack out programs and then get through compile and test with few defects. Their programs, however, are generally still defective. Our data show that the key is to get low defects by using sound practices. Then the products are generally defect free, test time is minimized, and the development process is more predictable. The advantage of the quality profiles is that they show both

the program's defect levels as well as the quality of the student's design and review practices. With these data, faculty can better advise their students on how to improve. For example

1. When components have high defects in test, they will generally have high defects in subsequent use [Kaplan 1994]. Test defects are thus a good indicator of the quality of a student's work.
2. When components have high compile defects, it indicates a poor-quality code review. This will also generally result in high unit test defects.
3. When components have low compile but high unit test defects, it generally indicates that students used poor design methods and did not spend enough time in the design review.
4. The amount of time students spend on design review and code review is indicated by the quality profile. When they spend a reasonable amount of time and still have many compile or unit test defects, they are using poor review methods.
5. Faculty must be cautious about using PSP and TSP data to grade any student's work. If students suspect that their grades might suffer if, for example they had more than 10 defects/KLOC in compile, everyone will have lower numbers. Use student data to coach but base grades on improvement and the accuracy and completeness of the data, not on the data values.

EDUCATION LESSONS FROM THE TSP

Perhaps the most powerful consequence of the TSP is the way it helps teams manage their working environment. A common software problem is unreasonable schedule pressure. While this is normal, it can also be destructive. When teams are forced to work to unreasonable schedules, they are unable to plan. As a result, they must work without the guidance of an orderly plan and will generally take much longer to complete the work than would otherwise be the case.

While software engineers will invariably face demanding schedules, history demonstrates that with effective planning and quality management, projects take less time. The TSP shows teams how to plan their work and it provides them the data to defend their plans. To date, most TSP teams have been able to convince management that their plans are aggressive but achievable.

The lessons we have learned from PSP and TSP are

1. When under severe schedule pressure, software engineers will generally concentrate on those tasks they know how to do, even if others are more important.
2. Thus, when engineers do not know how to develop firm requirements, produce a well-structured and documented design, or establish an effective working team environment, they will generally concentrate on coding and testing.
3. When given a clearly structured set of tasks and a defined process for handling them, engineers will generally follow the process. This assumes that they know how to follow a defined process.

We find that most engineers are willing to learn and, with proper coaching and support, they will use the practices they learn. On the other hand, if we want students to consistently use sound methods, we must provide them with far more explicit process guidance than we have in the past.

PSP and TSP Implications for Software Education

The principal implications from our PSP and TSP experience are that few engineers know how to estimate, plan, and track their work. Without these basic disciplines, they are unable to plan their projects and, without planning, they are generally under severe schedule pressure. Finally, when under severe schedule pressure, few engineers use effective design or quality practices.

Perhaps the most encouraging finding from our PSP and TSP work is that most software engineers recognize their current problems and are receptive to change. This is true for practicing engineers and also for those students who have some experience. One professor told me that one of her graduate students wanted to study the PSP. When she asked him why, he said "I want to see how good I am."

We have also found that without management and faculty support, even PSP-trained engineers will not continue to use PSP methods. Thus, to introduce the PSP, faculty need to understand and use it in as many of their courses as they can. The faculty themselves also need to use PSP methods both in grading students' work and in doing their own personal tasks. While this may seem like a lot to ask, students will want to know if their professors practice what they preach. Do they plan and track their work and do they focus on the quality of what they produce? For them to practice what we preach, it is important that we learn to practice it as well.

Many universities now offer graduate or senior level project courses. Again, unfortunately, these courses rarely provide specific guidance on how the students should run their projects. The emphasis is principally on producing running programs, not on how to plan, track, and manage the work. Because students in such courses often attempt too challenging a project, these courses are usually

painful lessons in how not to run a project rather than useful guidance on applying proper methods. While students undoubtedly learn from these experiences, there are so many ways projects can fail that learning through failure is not a viable educational strategy.

AVAILABLE PSP COURSE MATERIALS

The SEI has developed two university PSP courses: an intensive one-semester graduate or senior-level undergraduate course, and a course for freshmen [Humphrey 1995, Humphrey 1997]. The objectives, structure, and support for these courses are shown in Tables 4 and 5. These courses provide detailed guidance on how students should do their assignments. The course grading criteria focus on data quality and the proper use of the PSP methods.

The PSP should be first introduced to students in their freshman year. They will learn PSP planning and quality management and be able to practice these methods in their subsequent courses. For this to work, however, the students must use these methods in subsequent courses. This will not only help them build planning and quality management skills, but it will provide a measured framework for demonstrating the benefits of the methods they learn. For example, when students have data that show the numbers and types of design errors they typically inject, they can see how better design methods both reduce defects and save time.

TEACHING TEAM BEHAVIOR

With this foundation, students are ready to use the TSP on a project. This could be either a separate project course or a team project done as part of another course, say design. During the project, the students would regularly plan and track their work and report on progress. At the end of the project, each student team would analyze their data and write a project report. This report should include data on their work and a critique of the problems they found. These reports should include recommendations for what they would do differently next time and why.

THE PLANNED TSPE COURSE

Based on our experiences with TSP teams, we have designed and are now experimentally introducing a Team Software Process for Education (TSPe) course. This course is planned for either one- or two-semester dedicated project courses or for shorter projects in other courses. The TSPe course objectives are summarized in Table 6.

This course follows a somewhat different strategy from the industrial TSP. Since student teams are generally constrained to fixed schedules and resources, the course shows them how to plan a project to fit tight project constraints. It does this through a cyclic strategy where students build a first minimum-function version. Then, with the data on this initial cycle, they plan subsequent enhancements. Each cycle is expected to take from four to six weeks, so some courses could complete in one, two, or three cycles, depending on the semester length and the other course content.

Assuming the early tests with this course are successful, we plan to make TSPe more widely available for the 1999-2000 academic year.

CONCLUSIONS AND RECOMMENDATIONS

Based on our PSP and TSP experience, software engineers can be taught new methods and convinced to use them. The courses are challenging and require a lot of work but the resulting benefits are substantial. The graduate or senior-level undergraduate PSP course must be approached more as an experience than an intellectual exercise. Merely explaining the methods will produce little or no meaningful benefits. The students must work through the course exercises, measure their work, and analyze their data to see improved performance. When they do this, the course works. If they just read the text or listen to lectures, they are generally wasting their own and the professor's time.

The best evidence of how well the PSP course works is in how it changes the way students spend their time. As shown in Figure 6, at the beginning of the PSP course students spend less time on design than on any other activity, including compiling. At the end of the course, they typically spend more time on design than on coding, compiling, or testing.

In introducing the PSP, it is most desirable to teach the introductory course to freshmen with their first computer science courses. Then they should be required to continue using the PSP in the rest of their educational program. After a couple of years of experience, the students will have a substantial amount of data on their personal performance and will be proficient with the PSP methods. They would then be prepared to use the TSP team process in their more advanced courses.

While all the materials needed for the PSP courses are contained in the textbooks and accompanying support materials, we have found that some professors have trouble the first time they teach these courses. Generally, the reason is that the PSP emphasis differs substantially from traditional computer science and software engineering courses. To address this, the SEI and the Computer Science Department of ERAU jointly offer a one-week summer workshop on teaching the PSP and TSP.[2]

[2] Hilburn 1997.

The PSP also provides a substantial research opportunity. When students gather comprehensive data on their assignments, it is possible to conduct various software tool and method experiments or to try various teaching techniques. Another potential research area would be to identify language constructs and features that are most and least defect prone or language features that significantly impact engineers' productivity. One could also explore such topics as the nature and types of skills needed for software work and possible performance measures for these skills.

While the PSP and TSP are new, they provide an exciting opportunity for a quantitative approach to software engineering and computer science teaching and research. Consider, for example, track and field events; in most categories, world records are broken almost every year. While software has many important differences from athletic events, human performance has always improved when measured. There is no reason to believe that human software capabilities could not improve as well. If we knew how to measure and assess the key skills, we could find out. As one student said, he took the PSP course to see how good he was.

Acknowledgements

While it is impossible to thank the many people who have participated in this work, I particularly acknowledge Iraj Hirmanpour and the faculty of the Aviation Computer Science Department at Embry Riddle Aeronautical University. They were the first to adopt the PSP as a required graduate course in their Masters of Software Engineering curriculum and they provided invaluable guidance and support in developing and testing the freshman PSP course. They also launched and successfully completed the first TSP project with a graduate student team. I also thank Linda Parker Gates, Tom Hilburn, Soheil Khajenoori, Alan Koch, Jim Over, and Bill Peterson for their many helpful comments and suggestions.

References

[Ferguson 1997] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya, "Introducing the Personal Software Process: Three Industry Case Studies," *IEEE Computer 30*, 5 (May 1997): 24-31.

[Hayes 1997] Will Hayes, *The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers* (CMU/SEI-97-TR-001). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Humphrey 1989] W. S. Humphrey, *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

[Humphrey 1995] W. S. Humphrey, *A Discipline for Software Engineering.* Reading, MA: Addison-Wesley, 1995.

[Humphrey 1996] W. S. Humphrey, "Using a Defined and Measured Personal Software Process," *IEEE Software 13*, 3 (May 1996): 77-78.

[Humphrey 1997] W. S. Humphrey, *Introduction to the Personal Software Process.* Reading, MA: Addison-Wesley, 1997.

[Kaplan 1994] Craig Kaplan, Ralph Clark, and Victor Tang, *Secrets of Software Quality, 40 Innovations from IBM.* New York, NY: McGraw-HAcknowledgementsWhile it is impossible to thank the many people who ha issue and give an example of a large product where the correlation between development test and customer-reported defects was 0.964 with a significance of better than 0.001.

[Paulk 1995] Mark C. Paulk and others, *The Capability Maturity Model: Guidelines for Improving the Software Process.* Reading, MA: Addison Wesley, 1995.

Tables

Table 1. Acceptance Test Improvement

Not Using PSP	KLOC	Months Late	Acceptance Test Defects
1	24.6	9	NA
2	20.8	4	168
3	19.9	3	21
4	13.4	8+	53
5	4.5	8+	25
Using PSP			
1	22.9	1	1

Table 2. System Test Time Savings

System test time before PSP training			
	Project	Size	Test Time
	A1	15,800 LOC	1.5 months
	C	19 requirements	3 test cycles
	D	30 requirements	2 months
	H	30 requirements	2 months
System test time after PSP training			
	Project	Size	Test Time
	A2	11,700 LOC	1.5 months
	B	24 requirements	5 days

	E	2,300 LOC	2 days
	F	1,400 LOC	4 days
	G	6,200 LOC	4 days
	I	13,300 LOC	2 days

Table 3. Quality Profile Dimensions

Dimension	Meaning
Design/Code Time	The ratio of detailed design to coding time - when engineers do not take the time to produce a thorough design, they generally make more design errors. To reduce this risk, design time should equal at least 100% of coding time. /Ct >= 1.0: P1 = 1, Dt/Ct < 1.0, P1 = Dt/Ct
Code Review Time	The time spent in code review, compared with coding time - by doing a personal code review before they compile, engineers can find a large percentage of their defects. A thorough code review should take 50% or more of coding time. CRt/Ct >= 0.5: P2 = 1, CRt/Ct < 0.5: P2 = CRt/Ct
Compile Defects/ KLOC	The defects per KLOC found in compile - even with good review times and rates, the review still could have missed a lot of defects. For quality products, compile defects should be less than 10 defects/KLOC. CompD <= 10, P3 = 1, CompD > 10: P3 = 20/(CompD+10)
Design Review Time	Detailed design review time, related to detailed design time - a thorough detailed design review should take 50% or more of the time spent in detailed design. Anything less generally indicates an inadequate review. DRt/Dt >=0.5: P4 = 1, DRt/Dt < 0.5: P4 = DRt/Dt
Unit Test Defects/ KLOC	The defects per KLOC found in unit test - the number of defects found in unit test is one of the best indicators of the number that will later be found. When the unit test defects/KLOC exceed 5, subsequent problems are likely.

	UTD <= 5: P5 = 1, UTD > 5: P5 = 10/(UTD+5)

Table 4. The Graduate PSP Course

Objectives	1. To improve a software engineer's performance 2. To provide an understanding of what processes are, how processes work, and how a personal process can help the students do better work 3. To help engineers measure the quality of their programs and determine how to consistently produce high-quality programs. 4. To provide engineers with measures of their personal performance and benchmarks against which to judge their improvement 5. To show engineers how to make accurate plans
Prerequisites	1. Fluency in at least one programming language 2. A basic understanding of software design 3. Familiarity with basic mathematics through calculus 4. An appreciation of statistical principles is helpful but not essential 5. An awareness of formal methods is also helpful but not essential
Course Structure	1. The basic PSP course has 15 lectures of 50 minutes each. 2. An additional laboratory per week is generally helpful in assisting the students to complete their work properly and on time. 3. There are 10 programming assignments that average about 100 LOC each, although program 10 is generally a little larger. 4. Five report assignments require the students to analyze personal data. 5. Students generally take 4 to 6 hours to complete each programming exercise, one to two hours each for reports 1, 2, and 3, and about 6 to 8 hours each for reports 4 and 5.
Course Support	1. The course is fully described in my textbook: *A Discipline for Software*

	Engineering, Addison Wesley, 1995. 2. An instructor's guide describes lecture objectives, suggests grading criteria, comments on the assignments, and provides lecture overheads. 3. An instructor's diskette includes spreadsheets for analyzing student data and lecture overheads. 4. A support diskette provides data analysis and calculation support for the students.

Table 5. The Undergraduate PSP Course

Objectives	1. To provide beginning software engineering and computer science students with an appreciation of personal software engineering disciplines 2. To expose students to planning and time management methods 3. To provide working knowledge of basic software quality practices 4. To provide a disciplined learning framework that will counter the common hacker ethic that many new students develop
Prerequisites	1. The basic requirements for the CS1 and CS2 courses
Course Structure	1. The freshman PSP course is a companion to any other beginning two-semester pair of software or computer science courses that include writing 8 or more small programs. 2. The students do the regular course work, augmented with the PSP materials. 3. The PSP materials specify how the students are to do their regular work using disciplined methods 4. The PSP material takes about 6 lecture hours spread over the two semesters 5. It is suggested that these added lectures be given at the beginning of each of the two semesters.
Course Support	1. The course is fully described in my textbook: *Introduction to the*

	Personal Software Process, Addison Wesley, 1997. 2. An instructor's guide describes the lecture objectives, suggests grading criteria, comments on the assignments, provides a set of lecture overheads, and gives suggested quizzes and quiz answers. 3. An instructor's diskette includes spreadsheets for analyzing student data, lecture overheads, and quizzes. 4. A support diskette provides student data analysis and calculation support.

Table 6. The Planned TSPe Undergraduate or Graduate Team-working Course

Objectives	1. To provide computer science or software engineering students with practical experience working in a team development environment 2. To show such students how to plan and manage their own work in a team environment and under typical management conditions 3. To show students how to apply the knowledge they have gained in a practical engineering setting 4. To demonstrate the teamwork benefits of following disciplined individual methods 5. To learn how and when to ask for and give team support and assistance.
Prerequisites	1. Successfully completed either the freshman or graduate-level PSP course 2. The ability and an interest in working cooperatively with other students on a demanding project 3. Basic competency in designing and developing small to moderate-sized programs
Anticipated Course Structure	1. The course would be designed to support either a full one or two-semester project course or a shorter team project in another course 2. The students would typically work in 4 to 6-person teams. 3. The team process would start with an abbreviated launch workshop

	where the team would plan their project. 4. The course materials will provide several predefined optional projects, each of which could be completed in either the one- or the two-semester course format.
Anticipated Course Support	1. A textbook, including an academic version of the industrial-scale TSP process and exercise specifications 2. An instructor's guide to show a PSP-qualified faculty member or graduate student how to act as the team manager and coach. 3. Faculty and student data gathering and analysis