Software Process: Applying Industrial-Strength Methods in Engineering Education

Mark J. Sebern, PhD, PE
Milwaukee School of Engineering
sebern@msoe.edu
www.msoe.edu/se/

Abstract

Improving productivity and quality in software development is one of the major concerns of the software engineering discipline, as software systems grow to millions, and soon billions, of lines of code. Productivity and defect density levels that are considered very good today will be inadequate to keep up with this future growth. As a result, software development professionals and organizations are striving to improve existing processes and to develop new ones. One example of demonstrated benefits from these efforts is the success of the Personal Software Process (PSP) and the Team Software Process (TSP) developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. This paper discusses the structure of the TSP and PSP processes, industry experiences with their application, and their relationship to other process improvement frameworks, including the current version of the SEI's Capability Maturity Model (now known as CMMI). With this background, it reflects on the introduction of the PSP and TSP into software engineering curricula, and reports experiences at the Milwaukee School of Engineering, one of the first four ABET-accredited software engineering programs.

Introduction

With the introduction of baccalaureate software engineering programs around the world, and the accreditation of the first such programs in the United States, software engineering has become accepted as a full-fledged engineering discipline. This has occurred in the context of the critical importance of software to the global economy and society, and also reflects the role that software now plays in virtually all kinds of engineered systems. As with other emerging areas of engineering, software has had its share of difficulties and failures, but a growing body of knowledge provides a basis for current practice and for continuing future improvement.

As is the case in other engineering disciplines, software engineers must have appropriate knowledge and skill both in practice (what they do) and process (how they do it). Software engineering practice has many components, including requirements analysis, software architecture, design, implementation, verification and validation, application of formal and mathematical methods, adaptation to specific application domains, and underlying computer science foundations. Software engineering process addresses issues such as planning, estimation, quality management, teamwork, and continuing improvement of methods and techniques.

The purpose of this paper is to describe some areas of software engineering process, including approaches that have demonstrated effectiveness in industry, to report on how similar techniques have been successfully implemented in an undergraduate software engineering program, to reflect on some lessons learned, and to assist other software engineering educators in related efforts.

The Personal Software Process

The Personal Software Process (PSP) was developed by Watts Humphrey[6,10] at the Software Engineering Institute (SEI) of Carnegie Mellon University, based on his work with the Capability Maturity Model (CMM). The CMM, which has evolved into the Capability Maturity Model Integration (CMMI)[1], provides a framework for software development organizations to improve the quality of their processes and the resulting products. While the CMM and CMMI focus on the organization as a whole, Humphrey also wanted to scale software engineering process down to the level of the individual software engineer. His research convinced him that many elements of a highly mature software process could in fact be applied to individual work; the PSP is the product of that research and of the continuous refinement and evolution that has taken place since its introduction.

The key elements of the Personal Software Process are presented in Table 1.

Process measures	Base measures: effort, size, quality, schedule Derived measures: yield, productivity, defect density, etc.
Planning	Proxy-based size and time estimation Task and schedule planning
Quality management	Quality measurement and analysis Code and design reviews Design documentation and verification Defect prevention
Process improvement	Process improvement proposals (PIPs) Project postmortem, with data analysis and "lessons learned"

Table 1 Personal Software Process (PSP) Elements

In the PSP, effort is measured by the time expended in various process phases. For software development in conventional languages, the typical size measure is non-comment lines of code (LOC); to reduce variability in this size measure, coding and counting standards are employed. The primary PSP quality metrics are obtained by counting and categorizing defects, and tracking them by the process phases in which they were injected and removed. The time required to fix each defect is also recorded. From these direct measures, a variety of additional metrics can be derived, including productivity, defect density, yield from design and code reviews, and defect fix time by phase.

Product size and development time estimates are based on proxies. As in other disciplines, a proxy is a substitute measure, chosen because it is easier to estimate. For example, the floor area of a house is usually a very good indicator of the final construction cost, but it may be difficult for the prospective owner to visualize in advance. The number and relative size of the rooms in the house is easier to comprehend; if such a proxy can be reliably related to the final floor area, then it can provide a basis for effective estimation. In the initial PSP research, Humphrey found that program size, measured in LOC, was a good predictor of the total development time. He then developed a proxy based on the number and relative size of classes in a high-level conceptual design for the software product being estimated. This proxy-based PSP estimating process is referred to as PROBE. The relationship between the proxy measures and actual program size is established by analyzing historical data gathered as a routine part of the PSP process.

There are many aspects of software quality, but the PSP focuses on product defects. The rationale is that the time required to find and fix defects can easily consume a significant share of available resources, leaving little time to deal with other important quality concerns such as usability, compatibility, and performance. Process data permits the individual software engineer to assess product quality and the cost of finding and fixing defects. Design and code review processes and checklists are used to detect and remove defects early, generally at less cost than dealing with them in later test phases.

Since software processes are dynamic, an important component of the PSP is ongoing improvement. Problems are noted, and potential solutions are evaluated and incorporated. Each development project ends with a postmortem phase, in which the software engineer draws conclusions from the data and implements any needed process modifications.

The Team Software Process

While the PSP provides a framework for self improvement for the individual software engineer, much greater benefits are available when the disciplined software development process is scaled up to the team level. The Team Software Process (TSP) is designed to support self-directed teams composed of members who have skill and experience in executing disciplined personal processes. The integration of elements from the PSP and TSP is illustrated in Figure 1.

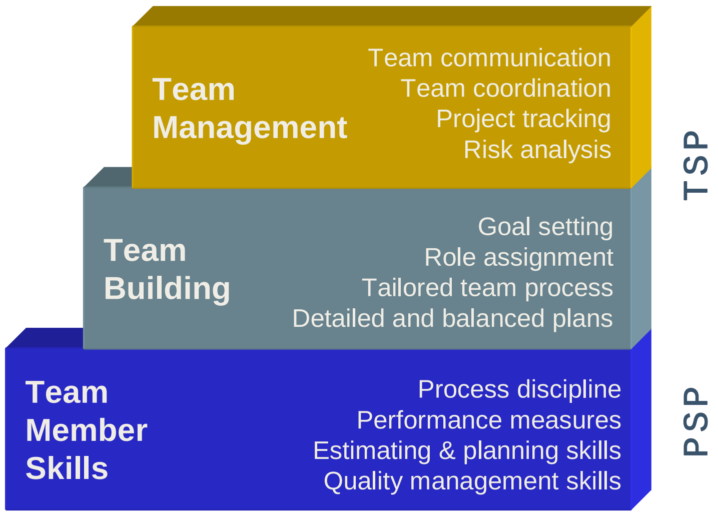


Figure 1. Elements of the PSP and TSP (from Davis[2])

The aim of the TSP is to create and support motivated, committed, and self-directed teams[9]. This can only be achieved when the individual team members have the necessary discipline and skills, but selecting a group of talented individuals is not enough. The TSP supports the team-building process that is necessary for the team to make realistic commitments and establish trust between the team and management. The primary team-building vehicle is the TSP launch process, which is organized as a series of meetings over a typical span of four days:

1. Establish product and business goals
2. Assign roles and define team goals
3. Define the development strategy and a tailored development process
4. Build top-down and next-phase plans
5. Develop the quality plan
6. Build bottom-up and consolidated plans
7. Conduct a risk assessment
8. Prepare the management briefing and launch report
9. Hold the management review

In the first meeting, senior management spells out the need for the project, how it supports the organization's long- and short-term goals, the desired delivery schedule, and possible tradeoffs between features, schedule, and cost. A major purpose of the meeting is to give the team the understanding and motivation needed to plan effectively and to evaluate alternatives.

In meetings 2 through 8, the team works with a coach to create one or more plans to achieve the management goals; these plans are presented to management in meeting 9. The team prepares a conceptual design and an accompanying development strategy, builds an overall project plan and a very detailed one for the next development phase (usually about three months), balances workload across team members, sets quantitative quality targets and a plan for achieving them, and identifies risks and ways to mitigate them. The result of this work is the team's most aggressive, realistic plan to meet the requirements articulated by management. If the resulting plan does not meet management's desired schedule, then the team prepares one or more alternative plans, using the understanding gained in the first meeting to make any necessary tradeoffs.

As part of the launch process, team members are assigned to specific roles, in addition to their general contribution to project tasks. These roles include customer interface manager, design manager, implementation manager, test manager, planning manager, process manager, support manager, and quality manager.

In meeting 9, the team presents its recommended plan to management. Management may accept this plan or one of the alternatives, work with the team to develop other options, or perhaps even decide not to proceed with the project. The desired result of this meeting is usually agreement between the team and management on a project definition and plan to which the team is committed and which meets the organization's needs. Occasionally, management is unable to make a decision and either asks the team for more alternatives or seeks other input.

Team-building can be a messy process, and a TSP launch is no exception, but the process of joint goal-setting, design, and planning provides a framework for effective team work. When a launch completes successfully, the team is committed to a project and plan that the members themselves have created, rather than one that has been imposed by management.

Of course, projects seldom proceed exactly according to plan, even when the plan is a good one, and this is where the team management component of the TSP comes into play. The detailed planning done during the launch allows the team members to start work immediately in a coherent and effective way, but it also provides a basis for replanning when changes in requirements or other events make it necessary to do so. Some people believe that projects fail because requirements change, and that any such changes should be forbidden. In the real world, though, requirements always change; trying to pretend that they don't only guarantees that the plan will quickly become out of touch with reality. Even if the requirements were to remain stable, software development is always a learning process, and the team must be able to exploit the increased understanding that is gained along the way.

For these reasons and others, it is a mistake to think that plans are static and immutable; as Watts Humphrey often says, "if you can't plan accurately, plan often." When requirements changes are requested, the team can with little effort determine their impact on schedule and cost, so that management can make informed business decisions about whether to stick to the plan or to make adjustments. When the development process reveals issues that necessitate modifications to the original design approach, the plan can be modified and the workload rebalanced. In this way, the team can maintain its commitment to a realistic plan and its motivation to deliver what it has promised. It is important for management to understand that TSP's dynamic replanning is not a way for the team to escape from its commitments, but rather a way to ensure success in spite of the changing conditions that are a part of any real development project.

To make dynamic planning possible, the team must track and analyze its own process data, as team members complete the fine-grained tasks that make up the overall plan. By using size, effort, and quality measures, it is possible to determine how the team's actual work compares to the current plan. Individual team members, or the team as a whole, can make adjustments on a weekly or even daily basis. Measures such as earned value (EV) and task hours permit tracking of task completion and the portion of the team members' time that is actually applied to project activities. (Many teams are surprised to find that, in a 40-45 hour work week, only about 15 hours are spent on planned project tasks, as distinct from other essential activities like meetings, training, and email.)

In a similar manner, the team also creates and tracks a quality plan. Using historical data, estimates are made of the total number of defects that will be injected during product development. Quality metrics like yield (percentage of existing defects that have been found by a given point in the process) and review rate assist the team in planning for appropriate quality assurance activities (e.g., personal reviews, inspections, testing). Tracking these metrics during development allows the team to determine, for example, whether there are problems that might result in a large number of test defects and excessive time in integration and system testing, so that corrective action can be taken early in the process.

PSP and TSP Results in Industry

All of this sounds good in theory, but obviously it is desirable to determine the effects of process changes in the context of actual software development projects. Unfortunately, there are a number of obstacles that complicate research in this area. Commercial and government organizations are often reluctant to publish data on their own projects and processes, or even to release this information for inclusion in studies conducted by third parties. In addition, there are a wide variety of confounding factors, including differences in organizational structure, target markets, application domains, and the details of previous software processes.

Nevertheless, some available data does help to characterize the benefits of applying the PSP and TSP in a software development organization. First, PSP training courses provide insight into the process performance of individual software engineers, before and after learning the PSP methods. The "before" data reflects the existing processes of these practitioners, while the "after" data suggests the type of improvement that the PSP can produce. Second, TSP teams gather data on their own performance as part of their normal development process, and some of this data has been published in summary form[2].

As outlined in Humphrey's original text[6], students in a traditional PSP training course write ten relatively small programs, while using a series of defined software processes that build incrementally up to the full PSP. Data on size and development time by program, for a sample of 810 software engineers[11], is presented in Figure 2. In some cases, one program is built by modifying or reusing parts of previous programs; when this occurs, only the "new and changed" code is counted. While there is considerable individual variation, the average program size is around 120 LOC and the average development time is about 5 hours per program.

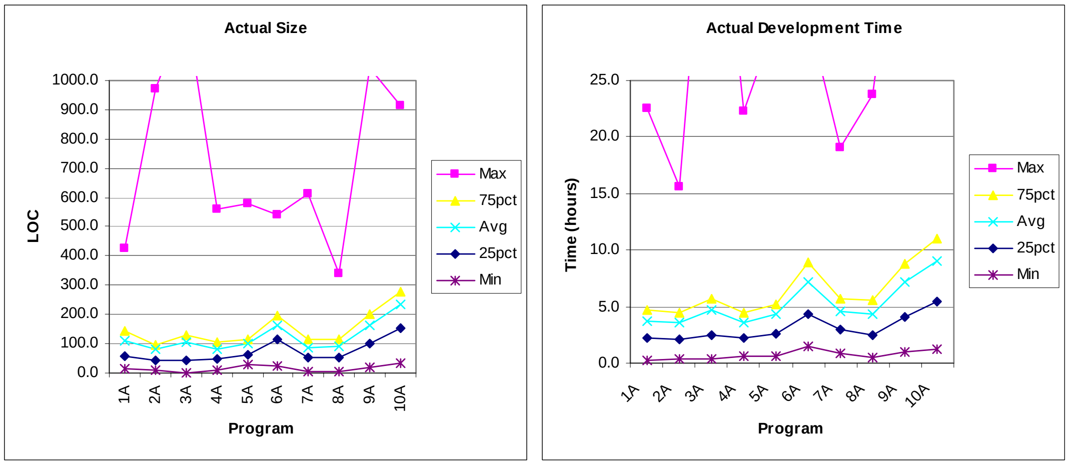


Figure 2. PSP Program size and development time (SEI).

The accuracy of size and time estimates for this sample is shown in Figure 3. The estimation error is calculated as

$$Error\% = 100 \cdot \frac{Actual - Estimate}{Estimate}$$

The vertical scale in these chart has been adjusted to show detail around the average values, with the result that many of the “maximum” error values (extreme underestimates) fall off the top.

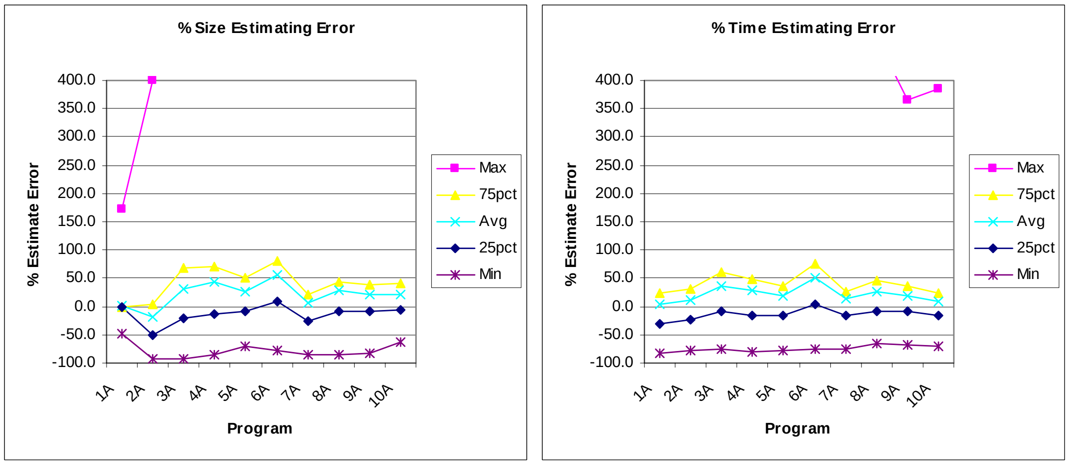
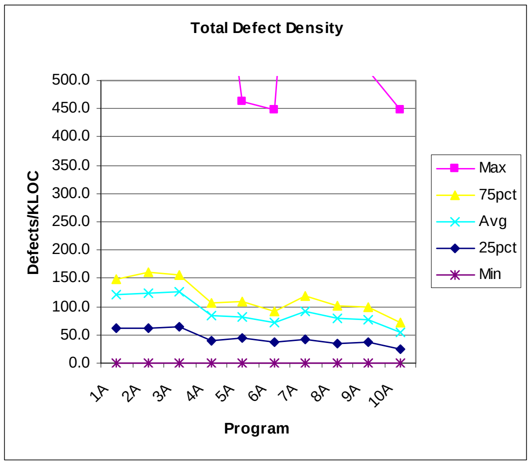


Figure 3. Size and time estimating error (SEI).

Although there are many derived quality measures, it is possible to gain some understanding by looking at total defects and defects found in the test phase. These data values, normalized to program size in KLOC (thousands of lines of code), are shown in Figure 4. (Note the difference in vertical scale when comparing these two charts.)



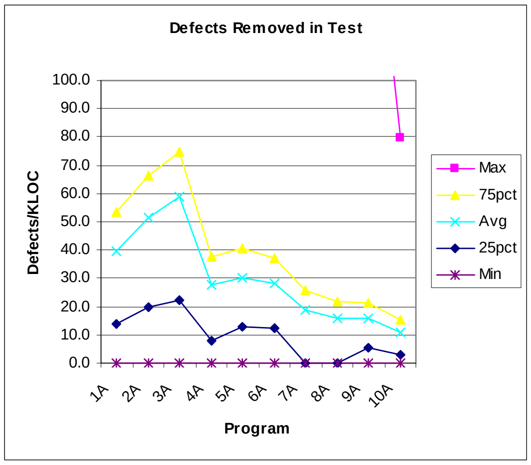


Figure 4. Summary of defect data, normalized to program size (SEI).

On the first program (1A), when the software engineers were using their habitual development practices, the average total defect density was about 120 defects/KLOC, or one defect for every eight lines of code. While this may seem high, it appears to be typical of practicing software engineers who are not yet using a disciplined development process. By the end of the 10-program sequence, using the incrementally enhanced PSP process, the average total defect density for this sample drops to a little over 50 defects/KLOC. Perhaps more importantly, the average normalized number of defects found in test, generally the most costly ones to fix, decreases from an initial value of about 40 defects/KLOC (and a high of almost 60 defects/KLOC) to a little over 10 defects/KLOC.

Another important quality measure is yield, which is the percentage of defects that are found by a specified point in the process. It is calculated by dividing the number of defects that have been detected at that point by the number of defects that have been injected up to that same point, and converting the result to a percentage value. Yield before compile is a primary measure of the effectiveness of the design review and code review. In the standard SEI PSP course, reviews are introduced with program 7A, which explains the shape of the yield curve shown in Figure 5.

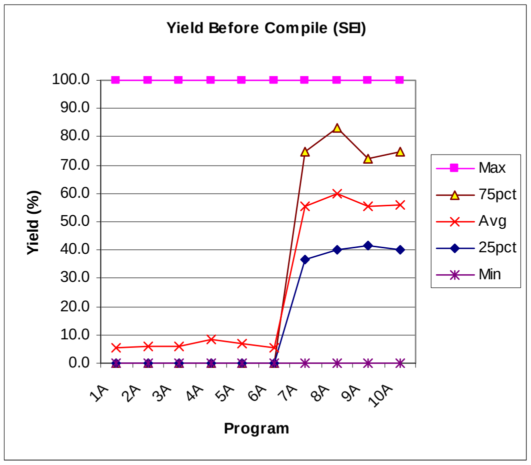


Figure 5. Yield (% of existing defects found) before compile (SEI).

An obvious question is how the added elements of the PSP process (e.g., planning, estimation, design and code reviews, and process improvement) affect productivity. By the end of PSP training, are these extra tasks offset by time savings from improved quality, or do they increase the needed development effort? For this sample, the result is shown in Figure 6.

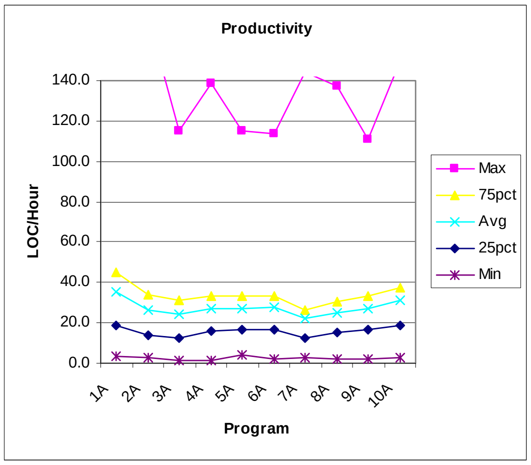


Figure 6. Productivity in LOC per hour of total development time (SEI).

While there is individual variation, the average productivity here remains just about constant. This means that the improvement in process predictability and product quality illustrated above has been achieved with little if any additional cost in overall effort.

Moving from the individual level to that of the software development team, a recent SEI report[2] analyzes data from TSP teams in thirteen organizations. The analysis results are presented in Table 2, and Table 3 summarizes related data from a recent industry survey[20] by the Standish Group.

	TSP Projects		Industry averages
Measure	Average	Range	
Schedule deviation error	+6%	-20% to +27%	(see Table 3)
System test defects (defects/KLOC)	0.4	0 to 0.9	15
Delivered defects (defects/KLOC)	0.06	0 to 0.2	7.5
System test effort (% of total)	4%	2% to 7%	40%
System test schedule (% of total duration)	18%	8% to 25%	40%
System test duration (days/KLOC)	0.5	0.2 to 0.8	N/A

Table 2. Summary of data from industry TSP projects (from Davis[2])

	Measure	1994	2000
Project resolution	Succeeded (time/budget/functions)	16%	28%
	Challenged	53%	49%
	Failed (canceled)	31%	23%
Challenged projects	Average time overrun	+222%	+63%
	Average cost overrun	+189%	+45%
	Required functions completed (%)	61%	67%

Table 3. Industry software project survey data (Standish Group International[20])

While the Standish report indicates improved schedule performance in the year 2000, it also notes that many information technology executives who completed the survey indicated that they routinely overestimate by 150%, extending the project duration by a factor of 2.5, when planning project timelines and making commitments. By contrast, TSP teams generally report setting the most aggressive schedules that they believe can actually be met.

In assessing the quality of products produced by TSP teams, it may be useful to compare these results with the average performance of software development organizations assessed at various levels of the Capability Maturity Model, as shown in Figure 7.

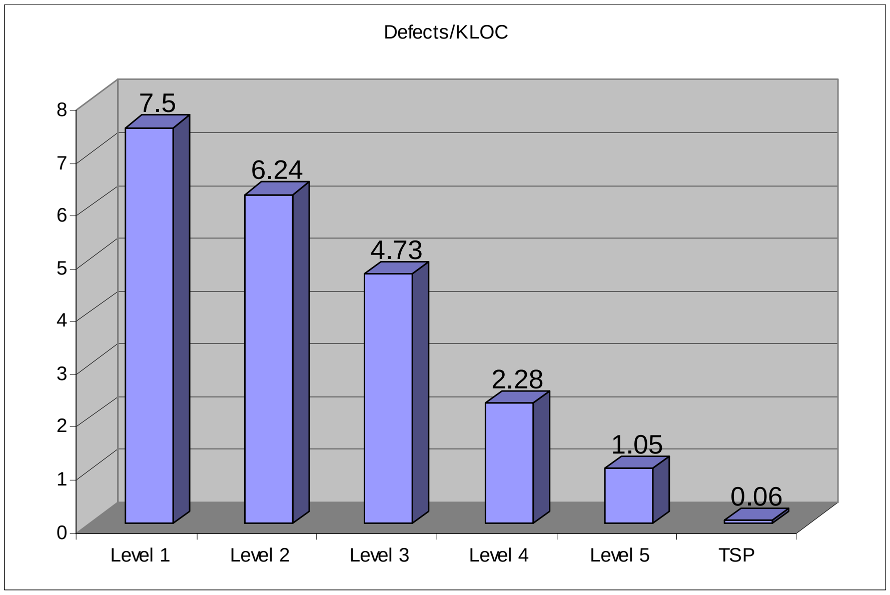


Figure 7. Average defect density of delivered software (from Davis[2], based on Jones[13])

The level of delivered product quality achieved by the TSP teams is significantly better (at 60 defects per million lines of code) than the norm for even "CMM level 5" organizations. Together with the planning results, this data suggests the use of the TSP as a mechanism for achieving high organizational maturity levels in significantly less time than the average for organizations that do not make use of the TSP. Indeed, one government organization[14] has reported moving from CMM level 2 to level 4 in 16 months, compared to an average time of 46 months.

In addition to the quantitative data, the SEI report by Davis and Mullaney[2] includes anecdotal comments from team members, which provide insight into the effects of learning the PSP and TSP and applying them to their development projects:

- "The best part about PSP/TSP is that collecting the metrics is for my benefit, not for someone else. I found that collecting the data proved to me that using a better process really does help my quality and productivity."
- "A more disciplined process allowed me to do a better job, and allowed me to balance my job with other aspects of my life."
- "Immediately after our launch, the team leader had a death in the family and needed to fly home. He was gone for two weeks. As a result of the TSP, the whole team fully understood what needed to be done on the project, and the team never missed a beat. Without the TSP launch, the team probably would have only been half as productive."
- "Our plans are much more detailed and all the involved developers understand them. As a consequence, we deliver what we planned, on time."
- "I feel included and empowered."

- "The first TSP team I coached was surprised when unit test was completed in half a day. They said they had done a prototype of this code before the project started and it took 1.5 weeks to get it to work well enough to see any results. They have found only two defects since the code has been integrated with the rest of the software."
- "Our project increased its delivered quality by 10 times and reduced its effort by 20 percent compared to a previous project."
- "This is the hardest, most enjoyable, personally rewarding thing I have done outside of growing a family."

PSP and TSP in an Undergraduate Software Engineering Program

The undergraduate software engineering (BSSE) program at the Milwaukee School of Engineering (MSOE) began operation in 1999 and produced its first graduates in May 2002. It was accredited by the Engineering Accreditation Commission (EAC) of the Accreditation Board for Engineering and Technology (ABET) in 2003, as one of the first four software engineering programs to be accredited in the United States. From the beginning, software engineering process was identified as a significant component of the curriculum, the current version of which appears in Table 4. MSOE's academic year consists of three quarters, each with 10 weeks of class and one week of exams. More detailed information on the MSOE SE program and curriculum is available at `www.msoe.edu/eecs/se/`.

Some computer science and software engineering programs[4] have attempted to introduce the PSP in freshman programming courses. However, others[15] have concluded that students need to establish a strong base of programming skills before trying to measure and improve their performance. For similar reasons, we chose to introduce the PSP in a sophomore course (SE280), after students have completed the freshman programming sequence and courses in data structures and unit testing.

Year	Fall	Winter	Spring
Freshman	CS1010 Computer Programming EN131 Composition GE110 Intro to Engineering MA136 Calculus I MS221 Microeconomics OR100 Orientation	CS1020 Software Design I EN132 Technical Composition HU100 Contemporary Issues MA137 Calculus II PH110 Physics of Mechanics	CH200 Chemistry I CS1030 Software Design II EN241 Speech MA231 Calculus III MA262 Probability & Statistics
Sophomore	CS2851 Data Structures MA232 Calculus IV MA235 Differential Equations PH230 Physics of Elect./Mag. SE2831 Intro to SW Verification	EE201 Linear Networks I EE290 Comb./Sequential Logic MA343 Matrices/Linear Prog. SE280 SW Engineering Process HU/SS Elective	CS280 Embedded Sys Software MA230 Discrete Math PH220 Phys. Heat/Waves/Optics SE2811 SW Component Design
Junior	CS3851 Algorithms CS386 Intro to Database Systems IE423 Engineering Economy SE3821 SW Requirements/Spec. HU/SS Elective	CS384 Design of Op. Systems OR402 Professional Guidance SE3091 SW Development Lab I SE380 Software Architecture HU/SS Elective	CS391 Embedded Sys Design HU432 Ethics SE3092 SW Development Lab II SE3811 Formal Methods Application Domain Elective
Senior	CS409 Ethical/Professional Issues SE4093 SW Development Lab III SE Program Elective Math/Science Elective HU/SS Elective Application Domain Elective	SE400 Senior Design Project I SE4831 SW Quality Assurance SE Program Elective Application Domain Elective Free Elective	MS442 Technical Management SE401 Senior Design Project II SS461 Organizational Psychology SE Program Elective HU/SS Elective

Table 4. MSOE software engineering curriculum (version 2.1)

Because of the limitations of a 10-week academic quarter, students in SE280 do not complete the full sequence of ten programming assignments recommended in the original PSP text[6]. Instead, this course requires seven programming assignments and three reports. Except for the reduction in the number of programming assignments, the first SE280 offering very closely followed the pedagogical approach recommended by Humphrey, progressing through the levels of PSP0 (baseline process), PSP1 (planning and estimation), and PSP2 (quality management and design). In this offering, students were also asked to manually record and analyze the process data.

This initial experience with the PSP went reasonably well, but several problems became apparent. First, perhaps in part because of the abbreviated schedule, the students found the frequent process changes to be very disruptive. Second, the manual data entry and analysis was time-consuming and error-prone, and tended to distract from the reflection and understanding that it was intended to facilitate. (PSP support tools have since been developed by the SEI and others[19], but were not available at the time.) Finally, the planning and estimation done early in the course suffered from a shortage of personal historical data.

To address these issues, a number of changes were made to the structure of the SE280 course[21]. Support tools (student and instructor spreadsheets) were developed to facilitate data recording and analysis, and to provide immediate feedback (through tables and charts) on individual and class performance. A single process, similar to PSP 2.0, was introduced at the beginning of the course and used throughout. Although detailed discussion of some process elements (e.g., design and code reviews) was delayed until later in the course, students perceived this as gradually deepening their understanding of a single process, rather than continually switching process frameworks. Coverage of estimating and planning was split into basic and advanced components. The basics of the proxy (PROBE) estimating method were introduced early in the course, so that students could start building a database of proxy ("class LOC") data from their conceptual designs, but the rest of the estimation process was deferred until the students' historical data was sufficient to support it. To accommodate this change in sequence, quality management topics such as reviews were moved earlier in the course; this had the added benefit of giving the students more experience with developing and improving their personal design review and code review checklists.

One concern we had before making these changes in SE280 was that we would lose the pedagogical advantage of having students see changes in their own performance as they moved through the process levels, discretely adding process capabilities at each step. Statistical studies[2,3] done by the SEI for the 10-program sequence do demonstrate significant differences between process levels. In our environment, however, the positive effects of the single-process approach seemed to outweigh this potential disadvantage. In part, this may be because sophomore students have not yet developed a lot of bad habits, and thus are more willing to accept what we present to them as an example of normal software engineering practice.

One benefit of the process support tool is that we have been able to gather historical PSP data on SE280 students, which can be compared with the SEI data presented above. After eliminating the students from the first course offering, as well as students who did not have complete and valid data for all seven programming assignments, this data set represents a total of 83 students and 581 assignments. Summaries of the results are presented in figures 8-12. The seven programming assignments reported here (1A to 7A) are the same as the first seven assignments in the SEI data although the development processes differ, as noted above. In the SE280 course, review phases are introduced right away, but detailed discussion of review techniques comes a little later, affecting the shape of the "yield before compile" curve (Figure 11).

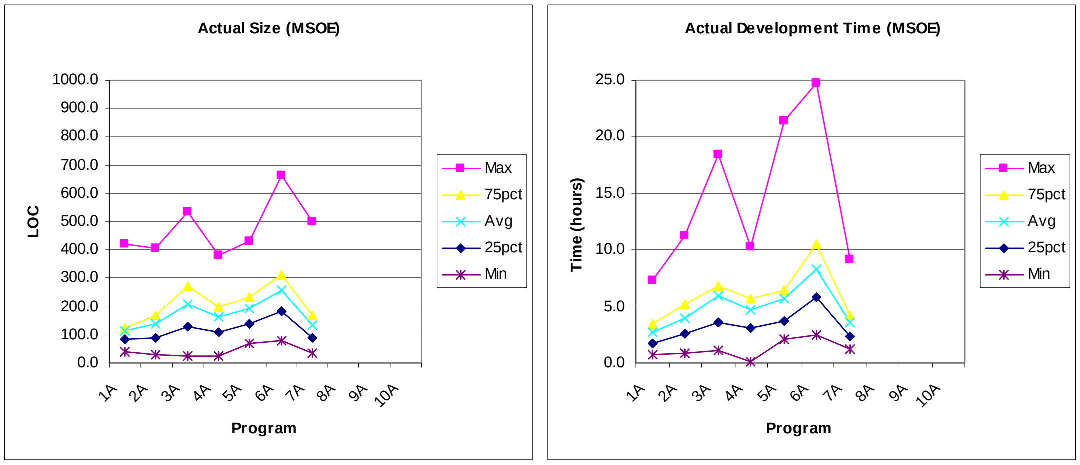


Figure 8. PSP Program size and development time (MSOE SE280).

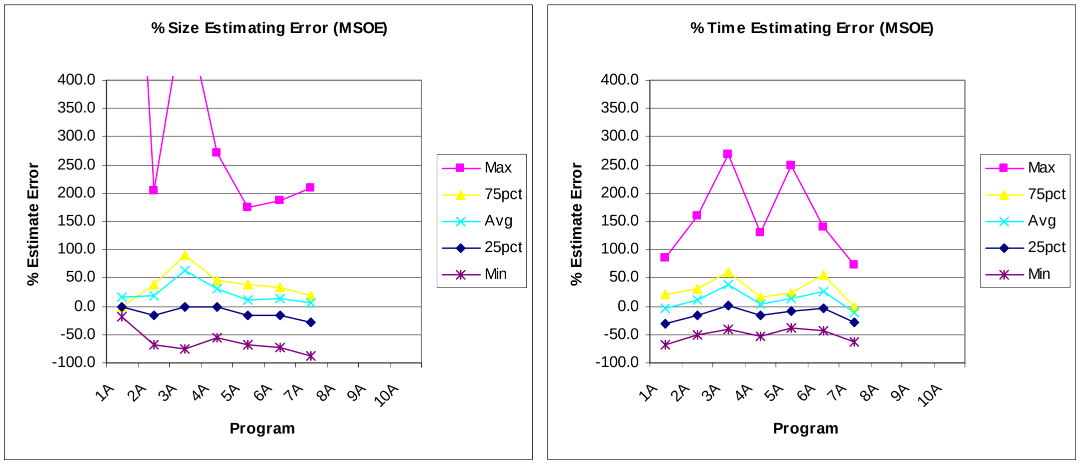


Figure 9. Size and time estimating error (MSOE SE280).

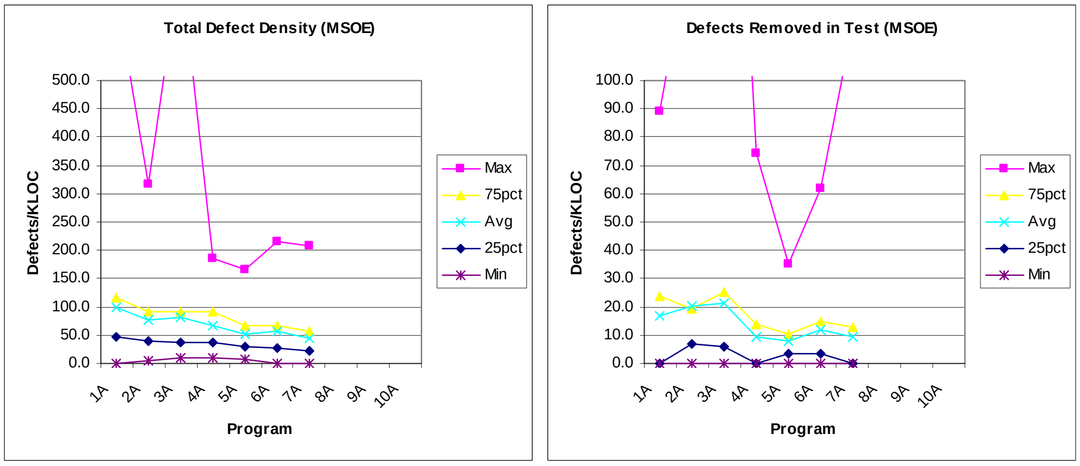


Figure 10. Summary of defect data, normalized to program size (MSOE SE280).

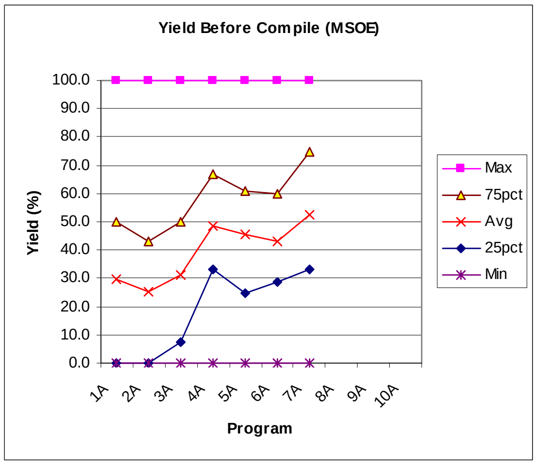


Figure 11. Yield (% of existing defects found) before compile (MSOE SE280).

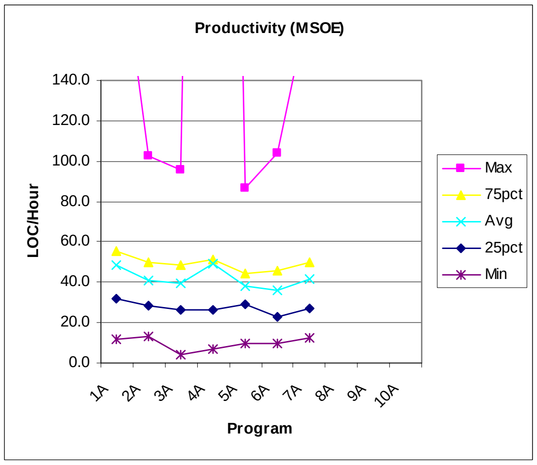


Figure 12. Productivity in LOC per hour of total development time (MSOE SE280).

Once students complete SE280, they are asked to use some components of the PSP on development projects in other courses, and some continue on their own to use the full process. One barrier at present to the greater use of the PSP in other courses is the lack of tool support that would permit students to maintain an ongoing database of process and project data, a problem that is currently being addressed.

Once students have learned how to use the PSP and to tailor and improve their own personal processes, they have the foundation necessary to work in a team process like the TSP. One approach to incorporating the TSP into a software engineering program is to dedicate a course to it, as in the case of the PSP. To meet this need, Humphrey has created an academic (or introductory) version of the TSP, known as TSPi, supported by a textbook[7] and other materials. It is intended to be used in a project course, but one where the projects are well defined and small enough in scale so that, as in the PSP course, the primary focus can be kept on learning and applying the process. The TSPi textbook provides two standard development projects, and warns against trying to substitute an open-ended, "real world" project within the extreme constraints of a single course. At MSOE, we have used this approach successfully in a single-quarter course on software engineering for computer engineering majors[16].

In MSOE's software engineering program, however, students are required to take a sequence of courses (SE3091, SE3092, and SE4093) in the Software Development Laboratory[17] (SDL), which provide an opportunity for a more in-depth experience in team process. This year-long activity involves working on large-scale projects for real clients, in teams, with defined processes and roles. The SDL development teams, generally with 4-6 members, have a dedicated work area in the lab that is available to them around the clock. The SDL process was initially based on TSPi, but has continued to evolve since the lab's inception in 2000:

- Cyclic development. As in the TSPi, SDL teams develop their products in successive cycles, with the goal of delivering a working product subset at the end of each cycle. This is an important motivating factor for the team members. Occasionally, it is necessary to spend the majority of a cycle on a single activity like requirements elicitation or testing, but these cases are the exception rather than the rule. In the requirements area, for example, it is important to strike a balance between having a sufficiently complete understanding of the system requirements and dealing with the reality that requirements always change during development, as both the team and the client deepen their understanding of the task at hand. The SDL teams make use of the TSPi "STRAT" form to document their current development strategy (e.g., what functions will be implemented in each cycle). In a 10-week academic quarter, most teams complete two development cycles, though there have been instances of one longer cycle or three shorter ones. One trade-off here is between minimizing planning overhead and retaining flexibility to adapt to needed changes in requirements, implementation techniques, or other factors.
- Planning. In the launch process at the beginning of each development cycle, the SDL teams review, and adjust as necessary, their project development strategy. They then make detailed plans for the tasks needed in the current cycle. The planning process involves balancing workloads across the team members, estimating the available task hours (time spent on development tasks, as distinct from other activities such as training, reporting, or staff assignments), and defining individual tasks with clear completion conditions. The size of these tasks is small enough to permit effective progress tracking.
- Team member roles. Specific responsibilities are assigned to individual team members, in addition to their common involvement as development engineers. At present, the SDL roles are:
  - Team leader
  - Development manager
  - Planning manager
  - Support manager
  - Configuration manager
  - Quality manager
  - Process manager
  - Requirements manager
  - Contact manager (client liaison)

  Each of these roles has defined responsibilities, documented on the lab web pages. The team member roles keep important team concerns from being neglected, while allowing each individual to make specific contributions to the team effort. Each role manager is empowered to call to the team's attention any issues related to his or her area of responsibility.

- Staff assignments. The scope of the TSPi process is limited to the individual development team but the SDL also needed a structured way to define, improve, and evolve the overall lab process. In the absence of such a mechanism, the development teams would address similar process issues, but not effectively share their experiences and results. For this reason, the lab process was modified to give students “staff” assignments in addition to their development responsibilities. While the staff teams vary over time, some common staff assignments have been the software engineering process group (SEPG), the software quality assurance group (SQA), and the training department. These roles are assigned to meet current needs and individual student interest; not all students are assigned a staff role in every academic quarter.
- Individual assignments. In addition to the ongoing team work, a small number of individual activities have been incorporated into the SDL courses. In the first SDL course, for example, each student chooses one CMMI process area and relates it to the SDL lab process, making suggestions for possible improvements. The intent of this assignment is to promote familiarity with both the existing SDL process and widely accepted industry practices.
- Student assessment. One of the difficulties in a project-oriented, team-centered lab sequence like the SDL is the assessment of individual students. Our techniques continue to evolve, but currently incorporate individual student portfolios with evidence drawn from project and staff activities, and tied to published course outcomes. Students are encouraged to spend a minimum amount of time on the portfolio submissions, but to plan their own work in such a way that the portfolio evidence is a normal byproduct of their task-oriented efforts. Anonymous team member peer evaluations are done periodically, with summary results provided to each subject student. The course instructor meets regularly with each team, and evaluations are solicited from project clients. The quality and completeness of process data is also a factor in student and team assessment. However, the actual values of process parameters (e.g., productivity or defect density) are not used for this purpose. As in the industrial TSP, the team’s data belongs to the team members and is used for their own planning and process improvement, not as a way to evaluate individual or team performance.
- Transition and turnover. The SDL course sequence begins in the winter quarter of the junior year and continues through the fall quarter of the senior year. With a few exceptions for “off track” students, this means that there is a 100% turnover in team members at the beginning of the winter quarter. In the fall quarter, the outgoing senior students are charged with managing a transition process to introduce the incoming juniors to the SDL. This process typically includes a series of presentations on the ongoing lab projects and an overview of the lab process. The incoming students express their project preferences, and tentative assignments are typically made by the midpoint of the fall quarter. These students are then involved in some activities with the senior team members during the rest of the quarter. In addition, each team follows defined exit and entry scripts to complete designated tasks before handing off the project and to assist the new team members in taking it over.

  We have considered arranging the curriculum to stagger the entry of new students into the lab, so that perhaps half of them would join in the fall quarter and the other half in the winter. However, the feedback from students to date has been that they would prefer to deal once with the transition difficulties rather than have the team composition change significantly for two quarters in a row. This may reflect the importance of a "jelled" team, a core TSP concept.

- Tool support. A web-based system supports planning and tracking; one of the lab's development teams is currently enhancing this tool, particularly in the area of quality management. An on-line "bug tracking" system is used to manage development issues, risks, and process improvement proposals (PIPs).

One difficulty with applying industrial-strength methods like the TSP in an academic environment is adapting to a typical academic schedule. While industry PSP teams have to allocate their time between the team project and other necessary activities, students in the SDL are typically taking four other courses at the same time. As a result, their time budget for the lab is about ten hours per week. Certain time-consuming TSP activities, including the launch process, have to be scaled down to reflect this reality and process overhead has to be kept to a minimum. Client expectations must also be managed so they are in line with the resources that the students are able to apply to product development.

Software engineering students at MSOE also complete a senior design project that is separate from the software development lab course sequence. These projects, which have a duration of two or three academic quarters, are often interdisciplinary in nature, with team members from different majors. Because of this, the students are not required to follow a development process that is identical to that used in the SDL. However, we have found that many senior design teams adopt variants of the SDL development process, even when software engineering students are a minority of the team members. Many of these teams use the same planning, design, and configuration management practices that are taught and applied in the software lab.

Although the evidence is largely anecdotal, a number of MSOE software engineering alumni have reported that prospective employers have been impressed by the depth of their software engineering knowledge in general, but specifically in the area of software process. Feedback from debriefing sessions with graduating seniors also confirms that students judge the software development laboratory and the senior design project to be among the most important components of the software engineering program.

The student and graduate feedback also suggests at least one area for potential improvement. Although students learn the PSP in their sophomore year, and apply a TSP-like process in the software development laboratory and senior design, it has been more difficult to integrate and reinforce these process elements in other courses. One difficulty, of course, is that in other classes the students are concentrating on learning other knowledge and skills, which impacts the stability of the students' development process. On the other hand, practicing engineers have to deal with this kind of problem every time a project adopts a new technology, and the PSP and TSP are designed to deal with this kind of process change. We are continuing to experiment with ways of further integrating software process, and the PSP/TSP in particular, throughout the software engineering curriculum.

Resources

For software engineering faculty members who would like to introduce the PSP and TSP into their programs, many resources are available. Some relevant books, reports, conferences, and tools are represented in the accompanying list of references. In addition, the SEI offers summer workshops for faculty members who would like to learn more about, and get practical experience with, the PSP and TSP processes. Academic pricing is also available for public PSP and TSP courses offered by the SEI. More information is available by following the training link from the SEI's PSP/TSP web page (`http://www.sei.cmu.edu/tsp/`).

Conclusion

Experience in industry has shown the SEI's Personal Software Process and Team Software Process to be effective methods for improving the software development process. A course on the PSP can be incorporated into an undergraduate software engineering curriculum with relatively little difficulty, and the results for engineering students are similar to those for practicing engineers. The TSP can be introduced in a stand-alone course or integrated into a more comprehensive software development laboratory, though some work is needed to tailor the full TSP to fit within the constraints of an academic schedule. Experience at the Milwaukee School of Engineering suggests that software engineering students are able to apply this process knowledge and skill to a variety of projects, and that these abilities are valued by employers.

Acknowledgment

Major support for the MSOE software development laboratory has been provided by Johnson Controls, Inc. This assistance, as well as continuing support from Johnson Controls management and staff, has made a significant contribution to the success of the software engineering program.

References

1. CMMI Product Team, *Capability Maturity Model Integration (CMMI), Version 1.1: CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SE/IPPD/SS, V1.1), Continuous Representation*, Technical Report CMU/SEI-2002-TR-011, Software Engineering Institute, 2002, http://www.sei.cmu.edu/publications/.
2. N. Davis and J. Mullaney, *The Team Software Process (TSP) in Practice: A Summary of Recent Results*, Technical Report CMU/SEI-2003-TR-014, Software Engineering Institute, 2003, http://www.sei.cmu.edu/publications/.
3. W. Hays and J. Over, *The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers*, Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, 1997, http://www.sei.cmu.edu/publications/.
4. T. Hilburn, "Software engineering - from the beginning", *Proceedings of the 9th Conference on Software Engineering Education* (CSEE'96), April 1996.
5. T. Hilburn, "Teams need a process!", *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE 2000), July 2000.
6. W. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, 1995.
7. W. Humphrey, *Introduction to the Team Software Process*, Addison-Wesley, 2000.
8. W. Humphrey, *The Team Software Process (TSP)*, Technical Report CMU/SEI-2000-TR-023, Software Engineering Institute, 2000, http://www.sei.cmu.edu/publications/.
9. W. Humphrey, *Winning With Software: An Executive Strategy*, Addison-Wesley, 2002.
10. W. Humphrey, *PSP - A Self-Improvement Process for Software Engineers*, Addison-Wesley, 2005.
11. W. Humphrey, Software Engineering Institute, personal communication.
12. Joint Task Force on Computing Curricula, IEEE Computer Society and Association for Computing Machinery, *Computing Curriculum — Software Engineering*, July 2004; http://sites.computer.org/ccse/.
13. C. Jones, *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, 2000.
14. L. Pracchia, "The AV-8B team learns synergy of EVM and TSP accelerates software process improvement", *Crosstalk:The Journal of Defense Software Engineering*, January, 2004.

15. P. Runeson, “Experiences from teaching PSP for freshmen”, *Proceedings of the 14th Conference on Software Engineering Education and Training* (CSEE&T'01), February 2001.
16. M. Sebern, “Iterative development and commercial tools in an undergraduate software engineering course”, *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 1997.
17. M. Sebern, “The software development laboratory: incorporating industrial practice in an academic environment”, *Proceedings of the 15th Conference on Software Engineering Education and Training* (CSEE&T'02), February 2002.
18. M. Sebern and T. Hilburn, “Integrating software engineering process in an undergraduate curriculum”, *Proceedings of the 18th Conference on Software Engineering Education and Training* (CSEE&T'05), April 2005.
19. Process Dashboard (software process support tool), http://processdash.sourceforge.net/.
20. Standish Group, *Extreme CHAOS*, The Standish Group International, 2001, http://www.standishgroup.com/.
21. D. Suri and M. Sebern, “Incorporating software process in an undergraduate software engineering curriculum: challenges and rewards”, *Proceedings of the 17th Conference on Software Engineering Education and Training* (CSEE&T'04), March 2004.
22. D.A. Umphress and J.A. Hamilton, Jr., “Software process as a foundation for teaching, learning, and accrediting”, *Proceedings of the 15th Conference on Software Engineering Education and Training* (CSEE&T'02), February 2002.

Biography

MARK J. SEBERN is a Professor in the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering (MSOE), and program director for MSOE's undergraduate software engineering program. He is also a visiting scientist in the Software Engineering Process Management group at the Software Engineering Institute and an ABET software engineering program evaluator.