

Teaching PSP: Challenges and Lessons Learned

Jürgen Börstler, *Umeå University, Sweden*

David Carrington, *University of Queensland, Australia*

Gregory W. Hislop, *Drexel University*

Susan Lisack, *Purdue University*

Keith Olson, *Utah Valley State College*

Laurie Williams, *North Carolina State University*

Watts Humphrey from the Software Engineering Institute at Carnegie Mellon University developed the Personal Software Process and first taught it as a graduate course at CMU in 1994. He says its goal is “to help you be a better software engineer ... As you study and use [PSP’s techniques], you will soon know how

to define, measure, and analyze your own processes.”¹ The PSP adapts a continuous improvement model to the specific needs of an individual software developer who wants to be more productive and produce higher quality software. In particular, the PSP targets the process used to individually design and develop software and incorporates ways to measure and change the process to achieve higher quality and increased efficiency.

Humphrey designed the complete PSP process as a semester-long university course for graduate students.¹ A student or professional learning to integrate the PSP into his or her process begins at Level 0 and progresses in process maturity to Level 3 (see Figure 1). During the course, each student accumulates personal historical data to create estimates and measure improvement. Each level incorporates new skills and techniques to improve software quality into the student’s process

and has detailed scripts, checklists, and templates to guide the student through the required steps. The measurement-based feedback in the PSP helps each student improve his or her own personal software process. Thus, Humphrey encourages customization of these scripts and templates as the students use this feedback to understand their own strengths and weaknesses. Students generally observe significant quality improvements as they progress through the four levels.²

Why teach the PSP?

The concepts that a graduating software engineer should know are well defined in the Stoneman version of the *Guide to the Software Engineering Body of Knowledge*.³ Of the 10 areas defined in SWEBOK, the PSP covers five: design, construction, testing, process, and quality. During a PSP course, students receive concrete experience in soft-

Software engineering educators must provide educational environments where their students can learn about the size and complexity of modern software systems and the techniques available for managing the difficulties associated with them.

ware metrics, life cycles, quality, and process improvement. The PSP lets students cover these topics both in theory and in practice.

With the increasing complexity of software projects, a growing emphasis exists on process maturity as a means of providing a quality product when time and budget constraints arise. The PSP provides a framework for high maturity processes scaled for an individual software engineer. Hence, it provides a meaningful way to instill process awareness in software engineers. Even if students don't use the PSP again, improving and making them aware of their programming habits will help them in their future academic and professional careers.

With the recent rise in popularity of agile software development methodologies⁴ (such as Extreme Programming), some educators might question the need to teach the PSP. Agile methods are highly incremental approaches that de-emphasize front-end analysis, design, and documentation in favor of communication and tacit knowledge transfer through human communication. However, Barry Boehm believes that both plan-driven methodologies (such as the PSP) and agile methodologies are important. "Both agile and plan-driven methods have a home ground of project characteristics in which each clearly works best and where the other will have difficulties. Hybrid approaches that combine both methods are feasible and necessary for projects that combine a mix of agile and plan-driven home ground characteristics."⁵

Teaching options

Depending on the environment, many ways to teach the PSP exist. The PSP material is quite extensive, so instructors might need to tailor and customize it to meet the needs of their class. We have identified three primary factors that influence the teaching of the PSP: course environment, coverage level, and support tools.

A PSP course's environment depends on the target audience, course level, and subject content. For professional software developers, teachers generally present the PSP in a distinct course using Humphrey's *A Discipline for Software Engineering*.¹ For students in computing programs, educators have tried integrating the PSP into almost every type of course from first to fourth year—as well as at the graduate level. Pub-

lished case studies show that teaching the PSP can be successful in any environment, but they also mention the difficult integration with first (CS1) and second semester (CS2) computing courses. Those who do integrate the PSP into CS1 often use Humphrey's *An Introduction to the Personal Software Process*.⁶ This shorter, easy-to-read book is specifically intended for teaching first-year students the basic principles of using disciplined software processes to produce high-quality software. In this article, we refer to this text as *PSP-lite*.

Another factor to consider when teaching the PSP is coverage of the material. In dedicated PSP courses, teachers introduce its four levels using 10 exercises that *A Discipline for Software Engineering* describes.¹ However, when integrating the PSP with other course topics, such a complete introduction is not feasible. Humphrey recognizes this problem and gives some suggestions for variations.¹ However, all the proposed variations are still staged introductions involving at least seven assignments. Instead, some teachers introduce just one specific (often slightly modified) PSP level and use it throughout the course, making it possible to reduce the number of exercises. The PSP can also be integrated into CS1 or CS2 using *PSP-lite*.

Having the proper tools to support the tasks involved in PSP related activities is also important. Manual data collection often leads to incomplete, inconsistent, or erroneous data sets. Many PSP studies mention this observation and identify tool support as an important issue. The instructor's guide accompanying *A Discipline for Software Engineering* provides spreadsheets to calculate statistics.¹ However, Philip Johnson and Anne Disney identify the main problem as student data collection, not teacher data collection and manipulation.⁷

University experiences

Using the three factors, environment, coverage, and tool support, Table 1 summarizes our experiences teaching some PSP variations at five universities.

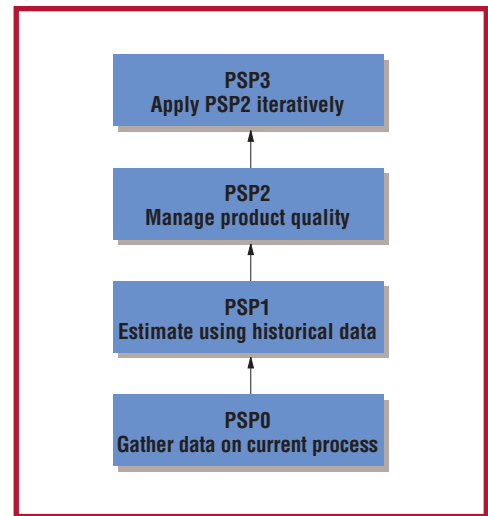


Figure 1. The levels of the Personal Software Process.

Table 1**PSP results at five different universities**

University	Environment	Coverage	Tool support	Comments
Umeå University	2nd-year C++ course 2nd/3rd-year SE	Modified PSP1 only	Locally developed tool	Optional usage was ineffective Team project (developing a PSP tool); good learning experience
University of Utah	CS1/CS2 Senior SE	PSP-lite across two courses Full PSP	Locally developed tool	Well received without much burden on students or teaching staff Students in pairs outperformed students working individually
Purdue University	CS1 only, CS1/CS2, sophomores	PSP-lite	Used provided spreadsheets	Students felt the data collection for the PSP was too much for an introductory programming course
Montana Tech	CS1 Junior CS/SE	PSP-lite Full PSP	Used provided spreadsheets	Taught as an add-on to CS1, exercises not correlated; Marginal success Students resisted data collection at first, but were very positive about the results at end of term
Drexel University	Graduate SE process course	Full PSP through PSP2	Standard forms	Very good learning experience although somewhat labor intensive for instructor and students

Umeå University

Umeå University used the PSP to exemplify disciplined software development. To minimize changes to the curriculum, we integrated certain basic PSP topics into two existing courses: a second-year C++ course and a second- or third-year software engineering course.

For the C++ course, we developed an extra-lite version of the PSP to deliver an essential PSP lesson—namely, “plan, track, and review what you did.” We introduced this adaptation of PSP in two extra 45-minute lectures and used it throughout all the exercises. We felt three medium-size exercises, with one or two weeks to complete each, were sufficient to collect historical data for trend analysis. We also provided a tool to support data collection and minimize data errors. PSP use was optional—only six of the 78 students used it throughout the course. None of these six students reported any perceived process improvements during the course. For the remaining students, the main reason for abandoning the PSP was the feeling that it would impose an excessively strict process on them and that the extra work would not pay off.⁸ Similar experiences occurred in other courses where PSP usage was optional. As a result, we do not advise optional PSP use.

In the software engineering course, we tried a different approach to familiarize students with the PSP. This course combines a group project and a theory track with lectures and assignments. During the theory track, we introduced the PSP as a bottom-up

approach to control software development, using published case studies extensively to show students that it was successful. During the group project, students developed tools to support data collection and trend analysis for the PSP. This forced students to actively acquire information about the PSP and its usage. Examination results showed that this teaching method helped the students understand the problems that PSP can solve.

University of Utah

For many years, teachers at the University of Utah used PSP-lite to teach the PSP to freshmen, integrating the material into both CS1 and CS2 and giving approximately five half-hour lectures on the material in each class. CS1 covers PSP-lite’s second half, which deals with defects (recording, reduction, prevention, and design-code reviews). Students estimate, track, and analyze the defects they remove from their programs. The book’s first half, which focuses on resource estimation and tracking, is taught in CS2, where the students estimate, track, and analyze the time they spend on their programs while continuing defect tracking and analysis. Teachers provide students with reports on class statistics and ask them detailed questions about their own PSP data on exams. Neither the teaching staff nor the students found integrating this material across the two classes to be difficult. Students reported that the knowledge gained on software engineering principles greatly aided them in obtaining summer internships.

Teachers taught the full PSP as well as a customized form of it that included pair programming during the last offering of a senior software engineering class in 2000. The university moved the software engineering class to the sophomore year and now relies on the PSP-lite taught in the freshman year. The class was a formal experiment to analyze the professed benefits of pair programming; students experienced even greater success when practicing pair programming while following PSP's practices.⁹ These students passed 15 percent more black-box test cases and spent approximately half the elapsed time when compared with students who worked alone following the PSP. Additionally, the pairs enjoyed themselves more, had higher confidence in their work, and encouraged each other to follow PSP practices.

Purdue University

Purdue University's Department of Computer Technology incorporated PSP materials into its introductory programming courses on two different occasions: once in the first course only and once across the two-semester sequence of introductory programming courses. Students in these courses were primarily sophomore computer information system majors who had already taken several courses to develop their computer literacy.

Faculty integrated the PSP-lite book and activities into the existing programming courses by covering half the book in each course. Students collected the PSP data as part of each weekly programming assignment. Because other topics needed to be covered, teachers devoted a limited amount of lecture time to discussing the PSP materials.

A course instructor checked the PSP data occasionally to make sure students were completing the activities correctly, and students completed a brief questionnaire at the course's end to rate their attitude toward the PSP. In general, the students viewed PSP activities as extra work added to the regular programming assignments and did not appreciate the potential benefits of a disciplined process. Susan Lisack reported that students made numerous errors on the PSP forms, and expressed negative attitudes toward the PSP on the course exit survey.¹⁰ In short, students felt they were already busy enough learning new language syntax and

the program development environment.

However, the students offered several suggestions to improve future course offerings. More automated tools would make it easier to record the required data, especially when more than one form needed the same data. The instructor should review student data early and provide immediate feedback about recording errors so that the collected data is meaningful. The instructor should also present actual data from the class to illustrate the PSP's benefits. Most importantly, students strongly recommended placing the PSP topics in a later programming course. Faculty responded by placing PSP topics into a graduate-level course on software processes and possibly including them in an upper-level undergraduate software methodologies course.

Montana Tech of the University of Montana

We also taught the PSP-lite and PSP in courses at Montana Tech. Our best success was with the full PSP, taught at the junior level. All students had completed two courses in data structures and algorithms and had a high level of competency in programming. The course met three days per week, and used the A series of exercises in Humphrey's text.¹ Student reaction was initially resistive, but in the end, the course received excellent student evaluations. The most common reaction was the students feeling more aware of their programming practices and shortcomings after the course. Faculty did not attempt to measure the students' performance in subsequent courses, but in senior exit interviews, the students all stated that the course made them more effective programmers.

Drexel University

The PSP continues to provide the focus for a graduate course in process improvement offered at Drexel University once or twice each year since 1996. The students in the course are masters students in both information systems and software engineering. The course addresses the PSP as presented in *A Discipline for Software Engineering*.¹ Recently, the course has also included some coverage of agile methodologies as a point of contrast.⁴

The course includes only seven PSP exercises because Drexel operates on a quarter term with 10 weeks of classes. This is sufficient for students to try the PSP through Level

The most common reaction was the students feeling more aware of their programming practices and shortcomings.

While understanding the PSP's mechanics is relatively easy, developing an appreciation for its goals and potential is much harder.

2. Drexel faculty has used both the A and B series of exercises, and has found that the B series works much better for the information systems students.¹ Teachers use the standard PSP forms and provide students with some example spreadsheets to demonstrate various PSP calculations such as regressions. Both instructors and students find the course work labor-intensive, but manageable.

Students in this course have a wide range of experience and programming ability. They include people with only minimal classroom programming experience, software developers with extensive experience, and managers who have not programmed in many years. These varying perspectives help enrich the discussion but also require the instructor to deal with each group's different problems approaching the PSP.

Overall results in teaching the PSP have been excellent. The faculty members teaching the course find it effective as a teaching vehicle. Some students are initially resistant to the PSP's requirements, and instructors have to be prepared to deal with these objections. However, by the course's end, students generally report that they found the experience valuable, and several of them report taking at least some PSP parts back to their work environments.

Challenges for students

The PSP is less about mechanics and more about instilling good habits and professional attitudes. As such, learners should apply PSP practices as a regular part of their studies, not just in a single course. Although widespread support is preferred, few universities are able to provide reinforcement throughout the curriculum because broad faculty support is required.

While understanding the PSP's mechanics is relatively easy, developing an appreciation for its goals and potential is much harder. For students without industrial experience, understanding the problems of large-scale, team-based software development is difficult. Student programs tend to be small and short-lived. Students experience neither the problems of post-release defects nor the value of high-quality software in production operation.

Students with industrial experience can have different problems. They usually

have first-hand experience with the key problems the PSP addresses. They understand why solving these problems would be good, but often they do not see the PSP as a workable solution. Many experienced students have ingrained programming habits and imagine that the PSP adds administrative overhead. Until they see the benefit for themselves, they can be resistant to it. In addition, these students might believe that a different way of working is impossible because managers will not support PSP practices.

Experienced or inexperienced students could also miss the PSP's value if they are unwilling to make a serious effort to try it. Students who simply do the minimum amount of work to get through the course are unlikely to gain any appreciation of the potential value that the PSP has for them personally as working software engineers.

Many universities teach software engineering practices that industry does not widely practice. This presents a particular problem for students when they move into the workplace. The typical situation is that a student joins a software development organization that neither knows about nor accepts and applies practices such as the PSP. Most students in this situation are likely to mold themselves to the local culture rather than using practices that their peers and managers might not respect.

Challenges and suggestions for teachers

With the rapid advances in computing technology, fitting the PSP into an already crowded curriculum is a continual problem for teachers. Universities are under increasing pressure to produce students with in-depth knowledge of the latest technologies for an industry with a chronic shortage of technical people. The mechanics of the introductory PSP are well within the reach of most students, although there is a need to devote teaching time to the PSP. The requirement for application of basic regression and correlation makes the full PSP difficult for first year undergraduates. However, it is reasonable to teach basic PSP processes in the early years, and then address topics requiring statistics in the later years.

Based on our experiences, we offer some guidance and advice to teachers interested in introducing the PSP.

Student motivation

Motivating students about the PSP's benefits is essential. Students enter a university with a predisposition toward coding without designing, and they do not see the data collection required for the PSP as enjoyable or useful. Teachers must overcome students' limited knowledge of commercial-scale software development and the work environment, and provide them with an appreciation for PSP's data collection, analysis, and measurement-based feedback. Introducing the PSP early may help students form good software habits, but teaching programming and the PSP simultaneously might cause cognitive overload for students.

The first half of PSP-lite deals with time recording, effort estimation, and commitment making, while the second half deals with defect reduction, prevention, and recording. Through the eyes of a beginning software engineering student, the material on defects in the book's second half seems more relevant. Even after just one program, many students have experienced the frustration of losing several hours to a small syntax or semantic error. They are motivated to reduce the frustration of defect elimination and are receptive to learning how to control their defects and minimize the amount of time spent debugging. We suggest that educators teach the material on defects prior to the material involving time recording.

Students are also more motivated to learn the PSP's lessons if the instructor can share stories of how previous students fared much better in job interviews because of their PSP background. Employers are impressed with students who can intelligently discuss the PSP and the principles it emphasizes.

Integration and adaptation

We have seen greater long-term success instilling the lessons of high-quality software development when teachers integrate the material across the curriculum.¹¹ Integration generally requires a commitment from multiple faculty members. It is not essential that these other faculty members inspect the students' time and defect logs, but these instructors can support the PSP's les-

sons by asking students to estimate, track, and report their PSP data.

A teacher's ability to adapt the PSP to the needs of their students is also important. Although it makes sense for experienced software engineers to develop the early PSP exercises in a waterfall-style process, this is normally not true for undergraduates who lack the programming experience to design and code even a small program in one iteration. Early on, teachers should show such students how to plan and perform multiple iterations. However, students should be discouraged from the practice of design-code-compile-test cycles at the single statement level. Apart from the practical problems of recording process data for such fine-grained iterations, such practices do not scale up to industrial-scale software development.

Dealing with data

The PSP lets students collect their own data to show how they've improved as they use better software engineering practices. When teachers reduce the number of exercises, students cannot see this progress as clearly because of a lack of data. Using published PSP data from other PSP users to talk about the PSP is possible, but it rarely has the same impact. The power of the PSP is when students realize that it works for them by seeing their own work improve.

Teachers must be able to process and present class data in a meaningful way. Students appreciate seeing aggregate class results on time estimation versus actual time spent (minimum, maximum, and average). They also like to see class quality trends (defects per KLOC) as they learn and apply new programming and quality techniques. To satisfy this desire for feedback, teachers need mechanisms for reliably and efficiently collecting PSP data from students and converting it into presentations. Ideally, these mechanisms would also let teachers provide individual feedback where appropriate and check the authenticity and validity of student data. Several tools are now available to support student data collection and analysis—for example, LEAP (<http://csdl.ics.hawaii.edu/Tools/LEAP/LEAP.html>) and the Process Dashboard (<http://processdash.sourceforge.net>). The challenge is to have tools that are convenient to use and don't distract from the work itself.

Teachers must also be conscious of the potential for misuse of PSP data. If students ever believe that this data is being used for grading purposes, they are likely to manipulate the values in an attempt to gain better grades. Students starting to use PSP often feel that there are ideal or target values that they should aim for. Teachers need to make it clear that the activities of collecting and analyzing the data are important, not the particular values.

We hope our experiences and experiments teaching the PSP provide guidance and advice for others interested in introducing the PSP into their software engineering courses. We continue to use elements of the PSP in our teaching because we believe that it is one of the few coherent and explicit approaches for teaching students about effective software engineering practices for developing high quality software. As with the PSP itself, the quantitative feedback available from teaching the PSP allows teachers to base their improvements on data, not just perceptions. ☺

Acknowledgments

This article is an outcome of the presentations and discussions presented at a workshop on teaching the PSP in universities held 20 February 2001 at the Conference on Software Engineering Education and Training (CSEE&T) in Charlotte, North Carolina (see www.spsu.edu/occe/cseet2001/registration.htm).

References

1. W.S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Boston, 1995.
2. A. Wesslén, "A Replicated Empirical Study of the Impact of the Methods in the PSP on Individual Engineers," *Empirical Software Eng.*, vol. 5, no. 2, June 2000, pp. 93–123.
3. A. Abran et al., *Guide to the Software Engineering Body of Knowledge: Trial Version*, IEEE CS Press, Los Alamitos, Calif., 2001.
4. A. Cockburn, *Agile Software Development*, Addison-Wesley, Boston, 2002.
5. B. Boehm, "Get Ready for Agile Methods, with Care," *Computer*, vol. 35, no. 1, Jan. 2002, pp. 64–69.
6. W.S. Humphrey, *An Introduction to the Personal Software Process*, Addison-Wesley, Boston, 1997.
7. P. Johnson and A. Disney, "A Critical Analysis of PSP Data Quality: Results from a Case Study," *Empirical Software Eng.*, vol. 4, no. 4, Dec. 1999, pp. 317–349.
8. S. Olofsson, *Evaluation of the PSP in the Undergraduate Education*, tech. report UMNAD 272.99, Dept. of Computing Science, Umeå Univ., Sweden, 1999.
9. L. Williams and R. Kessler, "Experimenting with Industry's Pair Programming Model in the Computer Science Classroom," *Computer Science Education*, vol. 11, no. 1, Mar. 2001, pp. 7–20.
10. S. Lisack, "The Personal Software Process in the Classroom: Student Reactions," *Proc. 13th Conf. Software Eng., Education, and Training*, IEEE CS Press, Los Alamitos, Calif., 2000, pp. 169–175.
11. M. Towhidnejad and T. Hilburn, "Integrating the Personal Software Process (PSP) across the Undergraduate Curriculum," *Proc. 27th Frontiers in Education Conf.*, IEEE Press, Piscataway, N.J., 1997, pp. 162–168.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

About the Authors



Jürgen Börstler is an associate professor of computer science at Umeå University, where he leads the software engineering and computer science education groups. Contact him at Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden; jubo@cs.umu.se.

David Carrington is a reader in the School of Information Technology and Electrical Engineering at the University of Queensland, Australia, where he is the program director for the software engineering program. His research interests include software engineering processes, methods and tools, and software engineering education. Carrington is also a visiting scientist with the Software Engineering Institute at Carnegie Mellon University. Contact him at davec@itee.uq.edu.au.



Gregory W. Hislop is an associate dean and faculty member of the College of Information Science and Technology at Drexel University, where he coordinates the college's software engineering and information systems programs. He has nearly 20 years industrial experience in software engineering and enterprise systems management. Contact him at hislop@drexel.edu.

Susan Lisack is an assistant professor in the Computer Technology Department at Purdue University, with interests in the areas of programming and databases. She holds certification as an Oracle8i Certified Database Administrator and also coordinates the departmental cooperative education program. Contact her at sklisack@tech.purdue.edu.



Keith Olson teaches at Utah Valley State College in Orem, Utah, where he coordinates the software engineering program for the Department of Computing and Networking Sciences. His principal research interests are in software processes and estimation. Contact him at olsonke@uvsc.edu.

Laurie Williams is an assistant professor of computer science at North Carolina State University. She received her undergraduate degree in industrial engineering from Lehigh University. She also received an MBA from Duke University and a PhD in computer science from the University of Utah. Her research interests include software development process, software testing and reliability, software security, and Ecommerce. Contact her at williams@csc.ncsu.edu.

