Experience Report: Teaching and Using the Personal Software Process (PSP)

Lutz Prechelt (prechelt@ira.uka.de)
Barbara Unger (unger@ira.uka.de)
Oliver Gramberg (gramberg@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/694092

Submission to ESEC 1997

January 16, 1997

Keywords: software process, software quality, metrics, measurement, process tools, empirical evaluation.

Abstract

PSP is a methodology for an individual software engineer's continuous self-improvement. Currently, few PSP experience reports are available from non-US sources, and hardly any from people other than the PSP inventor Watts Humphrey. We describe independent experiences with PSP. We find that PSP is a viable and useful approach and has quantifiable, positive impact. Problems in *teaching* PSP are in keeping students motivated and keeping them focused on general ideas instead of details. Problems in *using* a personal software process are keeping enough self-discipline and finding proper tool support.

1 The Personal Software Process (PSP)

The Personal Software Process (PSP) framework is an approach suggested by Watts Humphrey in 1995[1]. It describes a methodology that leads an individual software engineer towards disciplined, well-defined work with continuous self-improvement. The PSP ideas are independent of programming language, application domain, and team organization; they can be applied to programming as well as to many non-programming tasks.

Humphrey suggests to learn PSP in form of a 15 week course (effort: one 90-minute lecture and one exercise of 3 to 10 hours each week) that trains a set of techniques that form the basis of a personal software process. The student should then vary and optimize these techniques for his/her needs and introduce other techniques if required (therefore the name *Personal* Software Process).

The core ideas of the PSP framework are

1. to base the process on measurements, because "many people have feelings and opinions, but few people have data" (Humphrey) and
2. to make the process well-defined, because you can only improve what you do if you know what you do.

For further details see the appendix.

A problem with PSP (and the PSP course) as suggested by Humphrey is that it was more or less designed from the perspective of but a

single person, and not all of its aspects easily transfer to the needs of others. However, as far as we know only Humphrey has published PSP experiences in widely accessible publications, e.g. [2].

In the following we describe our experiences with teaching and using PSP, based on our courses for graduate students, researchers, and professional software engineers. There is one section on quantitative results with PSP, one on learning PSP and teaching the course, and one on using a personal software process.

2 Quantitative results

This section presents some quantitative results obtained in our first PSP course. These results confirm those published by Humphrey and add information about students' appraisal of the PSP course.

2.1 Student performance

The course performance of 20 university students is shown in the figures below. Figures 1 and 2 show the development of the defect densities over the 10 exercises of the course. We see in the thick trendline of Figure 1 that the total number of defects found during development per 1000 lines of code decreases significantly over time and the number of defects found late, namely in the test phase (Figure 2), exhibits a still more pronounced decrease. Figure 3 shows that the productivity is hardly impaired by the PSP during the course, despite the large fraction of bookkeeping effort involved.

But students do not only learn to produce software with less defects, they also learn to estimate more precisely how long it will take them to deliver the product. The absolute values of the time estimation error (in percent) are shown in Figure 4. The trendline in this case goes down from about 80% estimation error in the first exercise to about 30% in the last. As we see, average estimation errors are reduced significantly over the PSP course even though the process used keeps changing from one exercise to the next.

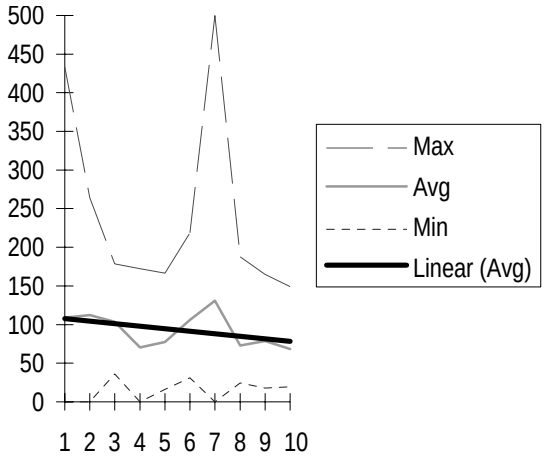

Figure 1: Total number of defects found per 1000 lines of code in each exercise for first course of university students. Left to right: course exercises number 1 to 10. Top line: maximum defect density of all course participants. Middle line: average defect density. Bottom line: minimum defect density of all course participants. Thick line: linear regression (trend) of middle line.

2.2 Course evaluation by the students

After this course, students answered a questionnaire with the following results (18 participants). The students spent between 4 and 20 hours per week overall for the course (average: 9.7). They judged the *difficulty* of the lectures slightly low on a scale from 1 (much too low) to 5 (much too high), namely 2.8 average, the difficulty of the exercises just right (3.1 average). On a scale from 1 (best) to 5 (worst), they found the course very *relevant* for their

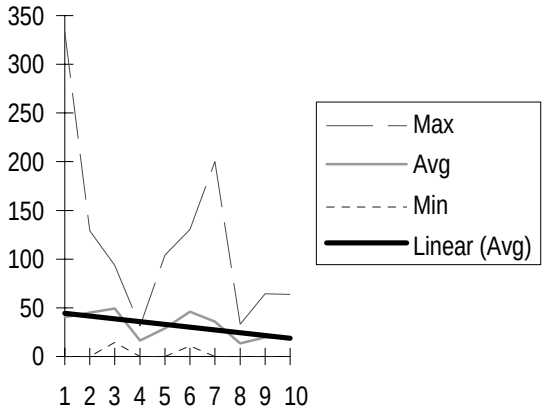

Figure 2: Number of defects found in test phase.

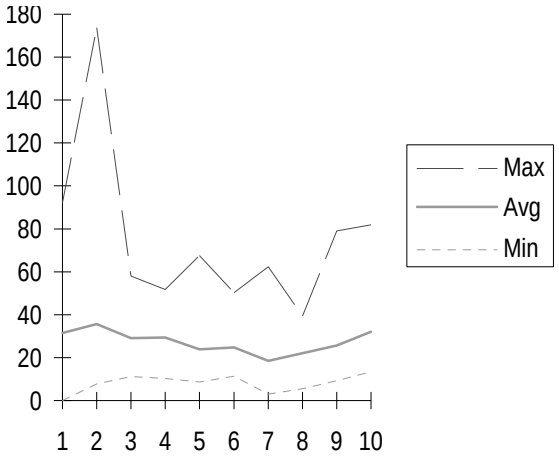

Figure 3: Total productivity in lines of code per hour over whole exercise.

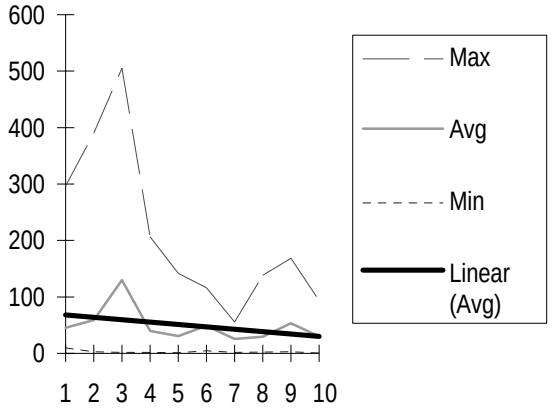

Figure 4: Time estimation error: $100 \cdot |T_{estimated} - T_{actual}|/T_{estimated}$

education (1.7 average), yet were only moderately well motivated (2.3 average). In the future, we hope to raise motivation by introducing reviews earlier. Overall, the course got quite high grades: 1.7 for the lectures and 1.9 for the exercises.

3 Learning and teaching PSP

After learning PSP ourselves (with a course group of five) we have now tought it four times: Twice to university students, once to junior university research associates, and once to professional software engineers. Two of these courses with 15 participants are still underway, 44 participants have already completed. This section reports our teaching experiences.

The university students were German informatics diploma students in their fifth semester or above (i.e., more than half through to the diploma). They have a solid theoretical education but typically rather limited programming experience. The research associates were all candidates for an informatics doctoral degree; they have a lot more programming experience from larger research projects. The professionals were junior and senior software engineers from a large software company.

We learned important lessons in three areas: providing and maintaining motivation for the course, keeping students focused on the important things, and properly running the course.

3.1 Motivation

The first half of the course deals mostly with estimating and planning, which compromises motivation: Subjectively, the amount of bookkeeping effort required for PSP planning appears unreasonable for two reasons. First, the fraction of bookkeeping effort *in the course* is indeed large, because the exercises are rather small. Second, at least students with little team project experience do not recognize why good planning is important at all.[1]

Therefore, we recommend to introduce reviews much earlier in the course than [1] suggests. Students then have the constant motivation to strive for zero defect programs and there is no longer emphasis on planning alone. In our next course we plan to introduce reviews in week 3 or 4 instead of week 8.

Another motivation problem regards taking the course at all: While our informatics students are highly motivated to attend the course, professional software engineers either are somewhat cynical and do not believe that *anything* could help them to improve their engineering practice, or they do not accept a course of 15 days length.

[1] As a result, students tend to personalize their personal software process in such a way that they "optimize away" several of its components and do not learn them at all.

To overcome this problem, we have developed a set of questions[2] which convinced every software engineer we have encountered so far that s/he has severe software process deficiencies. By linking the groups of problems addressed in these questions to PSP course contents, we can assure most programmers that a PSP course could help them. Overcoming the cynicism would require even harder proof of success of the PSP course than the metrics we currently supply. Therefore, we are currently defining a controlled experiment to compare the productivity and quality produced by PSP-trained students compared to non-PSP-trained students of comparable qualification. To reduce the course length we are currently developing a compact course of three modules of two days each.

3.2 Focus

The second significant problem in teaching PSP is that students tend to concentrate too much on the fine details of the individual methods suggested. For instance they concentrate so much on the questions which values of the β_0 regression parameter for time estimation are acceptable ones, that they do not understand why regression is used at all and which alternative methods are used when and why.

As some of the details are indeed complicated, we find it very important to keep the students' focus on the general ideas of PSP and on the general ideas of how to implement them instead of on the details of the specific implementation suggested in the course. On the part of the teacher, this requires in particular not to be picky judging the exercises, to emphasize the rationale of each method over its actual content, and to emphasize that all methods taught in the course are only suggestions and must be optimized based on personal data after the course. University students must be supplied with examples from industrial working conditions so they can understand the rationale of the methods suggested. Students that do not see the big picture will probably not be able to make improvements on their personal software process after the course.

3.3 Running the course

A few more remarks on how to run the PSP course: First, we found many of the PSP slides[3] to be too verbose for our teaching style and some slides contain material that the students already know — either from previous slides or from other courses.

Therefore, we have prepared a much different set of slides for our courses, waiving most redundant material. In particular, we combine the lecture pairs 3/4, 9/10, and 12/13 into one lecture each so that our course has three weeks without a lecture (but still with an exercise). These weeks can be used to compensate for holidays or to carry out extended discussion of experiences among the students.

Second, it is essential to provide sufficient room for such discussion in normal course weeks as well.

We reserve the first 10 to 15 minutes of each lecture for experience reports and feedback. We encourage students to tell success as well as failure stories. Most students find it very motivating to learn that their colleagues have much the same problems as they have themselves.

Third, this mutual motivation process works only if most or all students have already performed last week's exercise.

Therefore, it is important to enforce a relatively strict discipline with the time given for the exercises; all students have to deliver their exercise before the end of the course week, in contrast to the liberal delivery modalities most German students and professional software engineers seem to be used to.

[2] http://wwwipd.ira.uka.de/PSP/documents/questions.html

[3] provided by Humphrey on the teacher's disk, which is a supplement to the PSP book.

4 Using a personal software process

Although a personal software process is very useful in principle, its use is hampered by a number of severe problems. We discuss each of the most important ones in a separate section.

4.1 Lack of discipline

The single most important lesson we learned on using PSP is this: Properly using and improving a personal software process requires a lot of discipline; more than many people appear to be able to come up with.

Often, introducing appropriate PSP support tools (see also Section 4.2) will reduce the problem. For some people, the solution to the problem might be to find a group of colleagues that also use a personal software process. Such a group can keep up a process of mutual motivation, much similar to our PSP course groups. For others this might still not be sufficient. The key to successful PSP use for them might be to drop most of the standard PSP elements and use only what appears most useful for them. The latter is the mode of PSP use that all three authors apply 18 months after their own PSP course. For instance, we all do not use planning, because in a research setting this is rarely practical and often superfluous.

4.2 Tool support

As mentioned above, the bookkeeping required for measurements, gathering historical data, planning, and process improvement data analysis is a nuisance. Manual bookkeeping costs time, detracts from the main task, and provokes errors.

The bookkeeping issues to be addressed in a personal software process are the following:

1. making measurements (time per phase, product size, defects),
2. collecting and organizing historical measurement data,
3. analyzing historical data for process understanding and improvement,
4. computing plans and predictions from estimates and historical data.

Therefore, for industrial use of a personal software process, support tools are required. We have experimented with two measurement tools, several kinds of spreadsheets for data collection and analysis, and one specialized analysis program. You can find pointers to all these tools on the PSP resource page `http://wwwipd.ira.uka.de/PSP/`.

4.2.1 Titrax

Titrax (or TimeTracker, formerly known as *Timex*) is a simple X11 freeware timekeeping program by Harald Alvestrand; see the screenshot in Figure 5. The user defines a number of

Figure 5: Screenshot of Titrax timekeeping tool

activities and the program measures the number of minutes the cursor spends on each. The user has to click once each time s/he changes to a different activity. *Titrax* produces one small time sum file per day and can produce week-wise summaries. We have built an extension to produce summaries for particular activities or groups of activities over arbitrary periods of time.

In our experience, *Titrax* is a nice tool for general timekeeping but is not fine-grained enough for PSP use. Gathering defect correction time data with *Titrax* would be painful. Some of us use *Titrax* in their normal daily work but we all prefer *psplog* (see Section 4.2.3) for actual PSP time and error logging.

4.2.2 Spreadsheets

Humphrey offers a set of Excel spreadsheets on a support disk as a supplement to the PSP book. These spreadsheets support collecting and organizing the measurement data for the 10 exercises of the PSP course, but they are too inflexible and specialized for general professional PSP use.

We have written a large Excel spreadsheet that is more powerful and flexible; it supports parts of the estimation procedures and allows to select subsets of the data for use.

This spreadsheet can be a useful tool for professional PSP use in many cases. However, it is so large that it is difficult to master, and still is often not flexible enough or requires too much handwork.

4.2.3 psplog

In order to simplify the task of logging time and error data, we implemented a logging feature called *psplog* in the Emacs editor. It automates window switching and time stamp insertion on a single keypress. Figure 6 shows an excerpt from a psplog buffer.

This simple editor extension has proven a very helpful and practical tool. It is very flexible and sufficiently robust for professional use. The tight integration into the editor allows for accurate and most fine-grain time and error logging with a minimum of user detraction and effort.

4.2.4 evalpsp

To process the data gathered with *psplog* we have written a Perl script called *evalpsp*. It parses one or several psplog files and produces many sorts of statistics and summary tables in ASCII format, e.g. time spent per phase or per error type, number, percentage, or cost of errors introduced or removed per phase, etc.; 14 tables overall.

evalpsp makes the analysis of time and (in particular) defect data so simple that it becomes convenient to review such data for individual projects or in summary from time to time. The data produced by *evalpsp* is a good basis for finding candidate elements for process improvement. However, it would be helpful to get the consistency checking that is performed by *evalpsp* done during data gathering, i.e., integrated into *psplog*, as data gathering errors are the easier to resolve the earlier they are detected.

4.2.5 Tool perspective

The above tools provide reasonable support for data gathering and error data analysis. Less disciplined users will at times find parts of their gathered data missing or in disorder, because the tools can neither enforce correct data gathering (or any data gathering at all) nor do they perform early consistency checking. However, disciplined users will find the available tools sufficient.

However, the current tool support for handling historical time, size, and error data for PSP planning and estimating purposes lacks flexibility and integration with the data gathering tools on the PSP side as well as with project management and configuration management tools on the team side. It might be rather difficult to define a tool that is both adaptable to different personal software processes, yet still neatly integrated into larger software project tools. We are currently working on planning tools that provide at least part of such integration.

4.3 Personalization versus standardization

The final PSP problem concerns personalization. On one hand, adapting and fine tuning the PSP techniques to individual preferences and experimenting with new techniques will, for many software engineers, increase the power and quality of their personal process. On the other hand, such improvements not only consume precious worktime, with unpredictable payoff, but also reduce the ease with which a team can work together and can share data.

```
Jan  8 00:16:52 1997 bcr
Jan  8 00:18:16 1997 be 50 cd om  wrong order and names of Jbox() params
Jan  8 00:19:09 1997 ee
Jan  8 00:35:29 1997 be 80 ds om  forgot to assign xB = xBnew and yB = yBnew
Jan  8 00:35:49 1997 b
Jan  8 00:36:54 1997 e
Jan  8 00:37:09 1997 ee
Jan  8 00:44:47 1997 be 60 cd cm  paintBox() must only sometimes make a jump
Jan  8 00:47:21 1997 ee
Jan  8 01:11:28 1997 ecr
Jan  8 13:11:38 1997 bcp
```

Figure 6: Emacs psplog buffer with 11 event entries: begin of code review phase (`bcr`) on January 8th, 1997, 16 minutes after midnight, 3 begin error/end error pairs (`be, ee`), one interruption begin/end pair (`b,e`), end code review phase (`ecr`), and finally many hours later begin compile phase (`bcp`). The error entries are annotated with an error class, error insertion phase, error reason (omission, commission, education, communication etc.), and error description.

For instance if everybody has his or her own definition of what belongs into a design and what does not, it becomes more difficult for others to find design incompletenesses during a design inspection. It also becomes more difficult to predict implementation time from design time or design document size.

Therefore, the expected benefits from increased personalization of process elements have to be weighed against the benefits resulting from a single common definition of these process elements in a team. This is a tradeoff with no general solution.

5 Conclusions

Our experiences with teaching and using PSP can be summarized as follows:

- PSP is a good idea. Using a personal software process, software engineers can greatly improve the quality of their work and the reliability of their plans.
- However, for most people it is not easy to actually get PSP to work for them, mostly because of problems with self-discipline.
- There is currently insufficient tool support for using a complex personal software process. Missing tool support makes many useful process elements too difficult or too costly to apply.
- When teaching PSP, it is very important to keep the students' focus on the general ideas and to educate them to judge for themselves what is useful for them and what is not.

Appendix: PSP details

The PSP base techniques can be summarized as follows:

- Working in well-defined phases, e.g. plan, design, design review, code, code review, compile, test, postmortem.
- Measuring time spent per phase; protocoling time, cost, origin, and type of each defect made; measuring program size.
- Systematically collecting these data for future use.
- Systematically estimating product size from historical size data; estimating development time from product size; planning and tracking development schedules with clear milestones.
- Introducing design and code reviews[4] into the development process; systematically deriving checklists for reviews from historical defect data.

[4] These *reviews*, as opposed to *inspections*, are performed by the author of the artefact alone.

- Analyzing defect data to find opportunities to prevent defects even before reviews (quality management). Estimating number of defects to detect below-average quality.
- Introducing systematic design notations to improve clarity, consistency, and completeness of designs, regardless of the design method used.
- Systematically developing test cases and protocolling the test runs.
- Defining and documenting all process elements to improve consistency and to provide a precise meaning for all measurements.
- Introducing measurements to characterize process and product quality, in both predictive and explanatory manner.
- Introducing periodic, systematic process improvement efforts.

Acknowledgements

Thanks to Walter Tichy for commenting on the draft and to Stefan H"an"sgen for the initial psplog implementation.

References

[1] Watts Humphrey. *A Discipline for Software Engineering.* SEI Series in Software Engineering. Addison Wesley, Reading, MA, 1995.

[2] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.