Implementing Concepts from the Personal Software Process in an Industrial Setting

Khaled El Emam

Barry Shostak

Nazim H. Madhavji

International Software Engineering Research Network Technical Report ISERN-98-05.

Implementing Concepts from the Personal Software Process in an Industrial Setting

Khaled El Emam[a]
Barry Shostak[b]
Nazim H. Madhavji[c]

[a]Fraunhofer Institute for Experimental Software Engineering, Germany
[b]CAE Electronics Ltd., Canada
[c]School of Computer Science, McGill University, Canada

Abstract

The Personal Software Process (PSP) has been taught at a number of universities with impressive results. It is also of interest to industry as a means for training their software engineers. While there are published reports on the teaching of PSP in classroom settings (at universities and industry), little systematic study has been conducted on the implementation of PSP in industry. Also, largely anecdotal evidence exists as to its effectiveness with real programming tasks. Effectiveness is measured in terms of the number of trained engineers who actually use PSP in their daily work, and improvements in productivity and defect removal. In this paper we report on a study of the implementation of some PSP concepts in a commercial organization. The empirical enquiry method that we employed was action research. Our results identify the problems that were encountered during the four major activities of an implementation of PSP: planning, training, evaluation, and leveraging. We describe how these problems were addressed, and the general lessons learned from the implementation. An overall transfer of PSP training rate of 46.5% was achieved. For the engineers in our study, those who applied all of the taught PSP concepts on-the-job improved their defect detection capabilities.

1 Introduction

1.1 PSP Courses and their Effectiveness

The Personal Software Process (henceforth PSP) [14] is an evolutionary series of personal software engineering techniques that an engineer learns and practices. In its initial incarnation, the PSP has been intended as a classroom exercise course for graduate students or senior-level undergraduates. PSP courses have been introduced in a number of universities, such as Carnegie Mellon University [12], Embry-Riddle Aeronautical University [19], McGill University [34][35], the University of Massachusetts and Bradley University [15]. Moreover, it has been stated that both university students and experienced engineers gain substantial benefits from working on PSP [13]. PSP courses have also been used to train professional engineers in industry [12][13][23][33][26].

Existing measures of the effectiveness of PSP training are based on what the students do in the course exercises. Three of the more common measures of the effectiveness of PSP training are: (a) the proportion of students who actually complete the PSP course, (b) the extent of improvements in programmer productivity, and (c) the extent of improvements in defect detection and removal skills.

Completion rates obtained in classroom settings have varied. For example, a university PSP course had a completion rate of 53% (8 out of 15 students) [31], a completion rate of 4% (1 student in 24) for a course taught in industry [31], a completion rate from an industrial course of 45% (9 students completing the last exercise as opposed to 20 who completed the first one) [26], and the course completion rate of 77% (10 out of 13) [24] for training at Motorola.

In addition, it has been stated that students who go through the PSP course have average improvements in productivity ranging from approximately 21% [17] to 35% [12], and

"improvements of over ten times in the number of test defects" [12] and reductions of 58% in the average number of defects injected [17].

1.2 Effectiveness of PSP in Industry

With such impressive results, the PSP is of interest to industry as a means for training their software engineers. However, from an industrial perspective, the effectiveness of PSP should be evaluated based on what engineers do in their real programming tasks and in real programming environments, as opposed to course exercises and classroom settings. Improvements witnessed with course exercises in a classroom may not translate into equivalent benefits with real programming tasks and environments.

The difference between classroom and real programming tasks is highlighted by the potential confounding impact of reuse on the benefits of PSP in course settings. For example, the manner in which the PSP exercises are designed allows for substantial reuse from one exercise to the next. Sherdil [34] found that the extent of reuse in lines of code goes up as high as 75% for some exercises with an average of approximately 40% reuse. Such high levels of reuse can potentially explain the improvements commonly witnessed by students in terms of rising productivity and falling defect densities over time.

Thus far there has been only anecdotal evidence as to the benefits of PSP with real programming tasks [16], and only one recent report as to the number of engineers who actually continue using PSP concepts after the completion of the course (see [22]). If engineers do not continue using their PSP skills after the course then there is little motivation for management to support PSP courses.

As a preliminary exploration of PSP implementation in industry, we interviewed four ex-graduate students from McGill University who had taken the PSP course and who were currently programming in industry. We wanted to determine how PSP has influenced their personal processes. None of them was using the PSP techniques they were taught. It is therefore necessary to conduct a more systematic investigation of PSP in real programming environments to understand the barriers to the usage of PSP with actual programming tasks and to provide guidance for successful implementation in industry.

1.3 Overview of Paper

In this paper we present a study of the implementation of PSP concepts in a commercial organization. The empirical enquiry method that we employed was action research. Our results identify the problems that were encountered during the four major activities of an implementation of PSP: planning, training, evaluation, and leveraging. We describe how these problems were addressed, and the general lessons learned from the implementation. These results are hoped to be useful for other organizations embarking on the use of PSP as a vehicle for improving the processes of its software engineers. They are also a contribution to the research literature in terms of presenting an approach for implementing a process technology, and in identifying barriers to its implementation and strategies for overcoming them.

We conducted the study with 28 engineers. Briefly, our results indicate that 57% (16/28) of the software engineers who were taught PSP concepts (namely measurement and structured code reviews[1]) had the opportunity to apply the concepts in their real programming tasks. Also, 46.5% (13/28) of the engineers continue to apply the concepts in their programming tasks seven months afterwards. The engineers who applied code reviews witnessed substantial improvements in their defect removal capabilities.

The next section of the paper presents an overview of PSP so as to familiarize the readers with its basic concepts, and also introduces the context and objectives of our study. Section 3 is a description of our research method. In section 4 we describe the problems that we encountered in implementing PSP concepts, how these were addressed, and an evaluation of the implementation and its benefits. This is followed by the major lessons that we have learned during the implementation activities in Section 5. Finally, in section 6 we conclude with a summary of the paper and propose a framework for evaluating PSP implementations.

1 Of course measurement and code reviews have been used long before PSP came about, and so have many of the other PSP concepts. We are concerned here with the PSP packaging of software engineering concepts and educational material provided in PSP.

2 Background

2.1 The Personal Software Process

The PSP is intended to improve the personal practices of software engineers through the evolutionary introduction of good software engineering practices. These practices are scaled down versions of the applicable twelve practices from the CMM for software [13]. PSP is divided into seven distinct phases supported by 10 programming exercises.

In the first phase of PSP the students learn how to measure their work, as well as how to use the PSP forms and scripts. A predefined personal process is assumed that consists of the design, coding, compiling, and testing steps. These are preceded by a planning step and succeeded by a post-mortem step.

Subsequently, the students focus on size measurement, and size and resource estimation. A size estimation template is provided, and the students are taught techniques for size estimation, and applications in size and resource planning. The next focus of students is defect management. They are taught code and design reviews in order to increase early defect detection, as well as design specification and analysis techniques and the basics of process analysis. The final phase is a cyclic process which would help students scale up the skills that they have learned to larger programs.

2.2 The Study Context and Objectives

The organization with whom our study was conducted was CAE Electronics Ltd. located in Montréal and a leading supplier of flight simulators (henceforth referred to as CAE). CAE has a population of approximately 1200 software engineers. The Human Resources Department in conjunction with the site SEPG decided on a measurement-based process improvement strategy. The implementation of concepts from PSP was considered as part of this overall strategy.

At the first instance, it was decided that a pilot study should be conducted to: (a) tailor PSP to the context of CAE, (b) support a climate for change within CAE towards a measurement oriented culture, (c) evaluate the extent to which the PSP implementation approach has resulted in changes to the practices of the participating software engineers, and (d) evaluate the benefits of PSP within CAE (recall that much of the previously publicized benefits of PSP were based on the PSP exercises in classroom settings). Of course, evaluating new technology through pilot studies is a recommended practice during the diffusion of innovations into an organization [21]. In this paper we are concerned with this pilot study.

Planning for the pilot study started at the end of 1994, and the pilot was completed in February 1996. We had 28 software engineers taking part in this study. The details of the research method employed and of the implementation follow.

3 Research Method

The method that we have used to study the implementation of PSP is action research (e.g., see [3][10]). Following the action research method, researchers are involved directly in the introduction, observation, and evaluation of planned organizational change [3][2]. This is usually done as a collaborative effort with the sponsor of the change within the organization. Action research strives to achieve two objectives: (a) to solve practical organizational change problems, and (b) to increase our stock of scientific knowledge [32]. A more complete definition is given in [11] as follows (also, a characterization of action research in terms of factors such as internal and external validity is given by Jenkins [18]):

> *Action research simultaneously assists in practical problem-solving and expands scientific knowledge, as well as enhances the competencies of the respective actors, being performed collaboratively in an immediate situation using data feedback in a cyclical process aiming at an increased understanding of a given social situation, primarily applicable for understanding of change processes in social systems and undertaken within a mutually acceptable ethical framework.*

While no systematic investigation of the use and applicability of action research in software engineering has been conducted, we can draw some initial conclusions on its applicability by looking at the sister field of Management Information Systems (MIS). Traditionally, empirical MIS research has utilized four research methods [37]: (a) case studies, (b) field studies, (c) field tests, and (d) laboratory experiments. More recently, action research has been employed by MIS researchers and is considered a valid method to gain knowledge of relevant MIS phenomena [5]. Furthermore, if we consider our study to be on the implementation of

technology in an organization, then action research is considered an applicable research method [5][6].

4 Results

Similar to [23], we have broken up the PSP implementation study into four activities: planning, training, evaluation, and leveraging. These activities were not sequential. They do provide a useful way for grouping the issues that arose during the study. For each of these activities, we present the details of the implementation activity, the problems that were faced, and how these problems were addressed.

4.1 Planning

The purpose of the planning phase is to define the overall approach for the pilot implementation, recruit participants, and prepare materials (such as data collection forms and lecture slides).

4.1.1 General Approach for Implementing PSP

In training engineers on PSP, two primary options were considered: (a) classroom teaching and using the PSP textbook programming tasks, and (b) a mixture of classroom teaching and using on-the-job programming tasks. The former follows closely Humphrey's original prescription of teaching PSP in a classroom and giving the participants classroom exercises to practice the new techniques that they learn. The latter also uses classroom lectures, however, the participants apply the techniques they learn on their real programming tasks. In the current context, the former approach has a number of disadvantages as listed below:

- From our previous experiences teaching an earlier version of PSP at McGill University [34], delivering the full PSP took approximately 13 weeks of calendar time of 3 hours lecturing time per week. In addition, the students spent approximately 4.5 hours of effort per assignment (all of the students were experienced programmers with a median of 6 years of experience; however only 50% had industrial experience). This amounts to approximately a 70 hour commitment excluding time for feedback and for writing the reports. Humphrey notes that the full course takes between 150 to 200 hours of effort per engineer [17]. Such an effort investment was deemed too large since this time would have to be taken out of project schedules for multiple people working on the same project. This would have discouraged managers from supporting the training. This is important since supervisor support is an important determinant of positive transfer of training skills to the job.
- Four professional software developers who had previously taken the PSP course at McGill University were interviewed to determine how PSP has influenced their personal processes. None of them were using the PSP concepts in their real programming tasks. Stated reasons for not using the PSP concepts in practice included:
  - Organizational programming tasks, as opposed to the PSP classroom programming exercises, are usually conducted in teams. It was not obvious how the personal tasks of the PSP could be applied to team-work.
  - PSP requires the collection of a substantial amount of data. Without automated tools to manage and interpret this data, it could become cumbersome to apply PSP. The appropriate tools were not available to these individuals in their organizations.
  - There was a lack of management support. The overhead of data collection and interpretation in the short term will have to come from their own time unless management sanctions it. They are already putting overtime, so they do not really have their own time for PSP anymore. Management support is necessary for maintaining skills that are learned during training [27][8].
  - There is a hero culture in the organizations they work in. Those programmers who have a disciplined process and who produce code with less defects are less respected than the troubleshooting heroes who save projects in crises due to inadequate programming practices. This means that the current reward structure of the organization is incongruent with the principles of PSP, which makes it extremely difficult to practice PSP techniques on the job (e.g., see [28]). Therefore, there is no motivation for a disciplined programming process.
  - It is difficult to implement the disciplined practices of PSP alone when the rest of your team are not doing it. A lack of support

from peers is believed to inhibit practicing PSP on the job (e.g., see [28][27]).

Therefore, it is evident that there are practical problems transferring what has been learned in the university classroom to actual programming tasks and environments.

- It is believed that, in general, only 20% of critical job skills are learned from formal training and education, the remainder are learned on the job [7]. Therefore, there is a case for providing the participants with on-the-job training as part of the PSP implementation.
- We wanted to evaluate the benefits of PSP concepts in the real working environment of CAE, and not in an artificial setting. Therefore, we had to evaluate the benefits of PSP concepts on actual programming tasks.
- We wanted to gain information for tailoring PSP to the working patterns of CAE. Therefore, it was necessary to study PSP with engineers working on real tasks, not on classroom exercises.

The choice made was thus to have a mixture of classroom lectures followed by on the job programming tasks.

4.1.2 Selection of Participants

Ideally, the participants in the pilot study would be selected randomly from the population of engineers at CAE. This would ensure that the sample taking part in the study is representative of the population to whom we want to generalize. In field settings this is frequently difficult to do. Even though a truly random sample was not possible, we attempted to recruit participants from a diverse number of departments and businesses within CAE so as to ensure a reasonable level of representativeness. In total, we had participants from seven different departments.

One other important criterion that was taken into consideration during participant selection was that we wanted to select participants from the same teams to take part in the study. This will ensure that each participant's immediate colleagues are also learning and using PSP, and hence providing mutual support.

4.1.3 Design of the Study

One of the objectives of the pilot study was to evaluate the benefits of following the techniques of PSP. Therefore, embedded within the PSP implementation was a research design to evaluate the PSP concepts.

For the evaluation we used the general quasi-experimental design shown in Figure 1. The notation that we use to describe the design is as follows: an "O" represents an observation/measurement of a dependent variable, and an "X" represents an intervention. Also, the horizontal axis represents the time dimension. The same design can be extended to take into account more than two interventions. This design was used because we expected that the period between times t_0 and t_1 would be necessary for the participants to use and customize their data collection forms.

At the beginning of the evaluation (time t_0) the participants received intervention X_1. This is the module of PSP that covers basic measurements of the personal software process. Subsequently, the participants developed real programs on the job, and collected the relevant PSP data at the same time.

After five months, the participants received the second intervention X_2. This is the code review lecture. The reason we chose code reviews was that we wanted the second intervention to be "dramatic" so as to achieve buy-in into PSP by the participants and management. Humphrey states that code reviews should bring dramatic positive benefits to the personal process [14].

One issue that needs to be considered is the motivation of the participants. All the participants were volunteers, i.e., none was required to attend the PSP course or use PSP as part of their job.

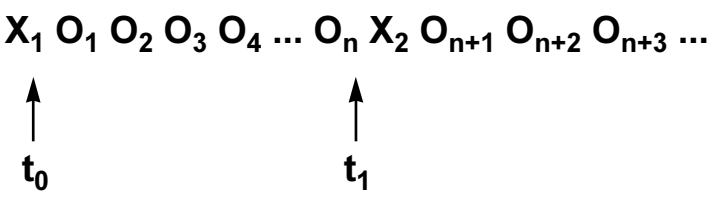


Figure 1: Design for evaluating the benefits of PSP.

When the participants were interviewed at the beginning of the study about their motivation, they all indicated that they believed that the PSP techniques would improve their personal processes (based on what they had heard and read). Of the 28 participants, 24 had a desire to improve their practices, 3 were interested in quantifying their work, and one was interested in "helping out with the study".

In order to control the transfer of skills to the participants, we had to ensure that they did not have access to material for the second intervention (or any other PSP material that may influence their practices). This was achieved by only giving the participants the relevant parts of the PSP manuscript that we wanted them to have. For example, at time t_0 the participants were not given the PSP material on code reviews.

4.1.4 Definition of Programming Tasks

For our approach to work we had to define a programming task that is akin to PSP exercises. We found that a universal definition for all participants was not possible. We therefore interviewed each participant at the beginning of the study and asked them about their current personal processes. During this interview, and based on their current personal processes, we defined for each interviewee what is a programming task. For example, for some engineers a task would be developing a module that takes less than two weeks of calendar time (which may necessitate decomposing larger modules); for others it was responding to a single change request.

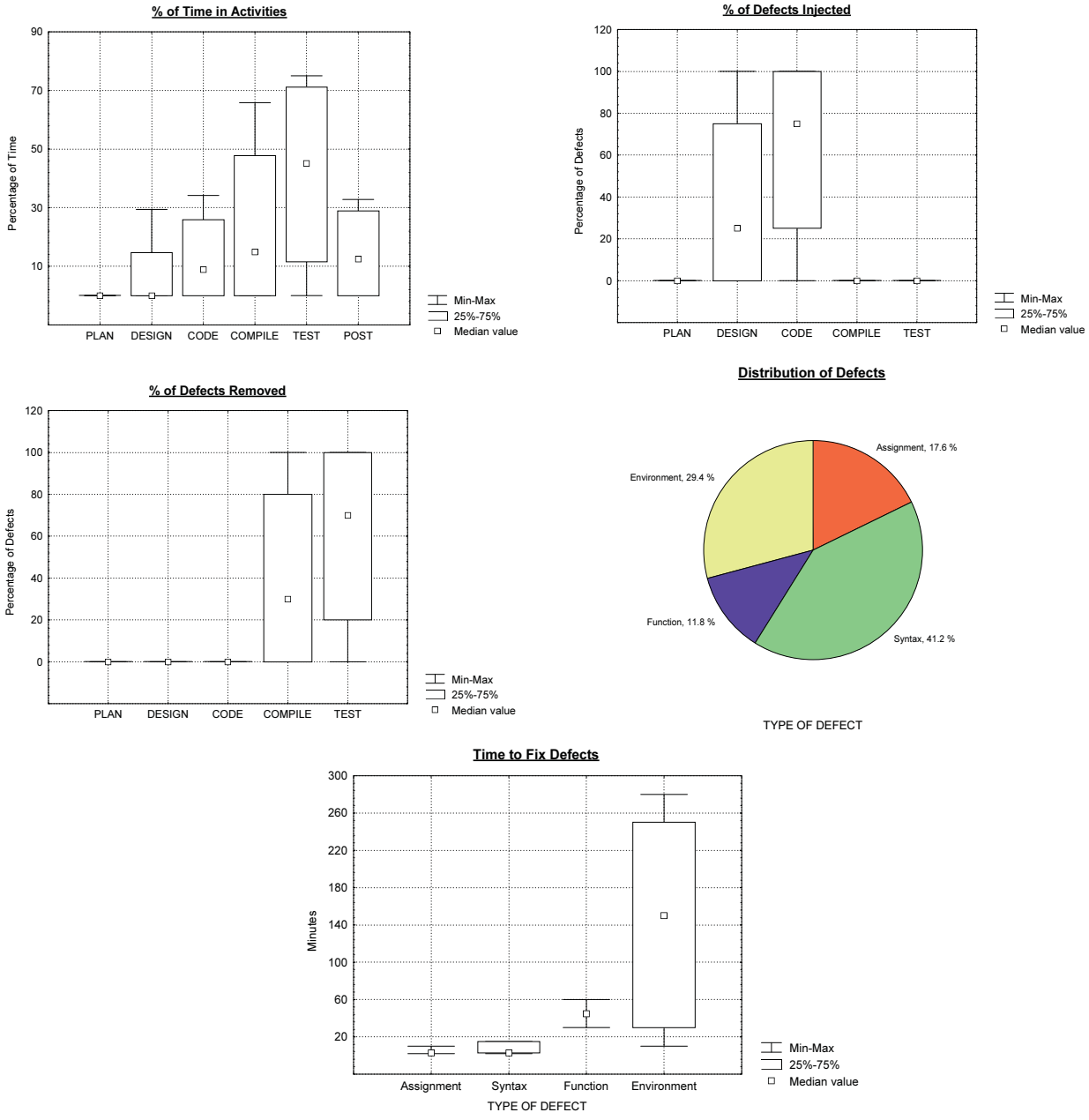
One problem that did occur was "blocking". This happens when progress on a program stalls because a needed resource is missing. This sometimes caused programs to be swapped in mid-stream. In some cases, programs were not completed before the end of the study for that reason.

Blocking was a key factor resulting in different engineers completing a different number of programs during the study. The number of completed programs per engineer was as low as one and as high as ten for the duration of the study.

4.1.5 Modification of the Data Collection Forms

Design of the data collection forms is perhaps one of the most important considerations in a PSP implementation exercise. While the PSP text does provide forms, difficulties did occur when attempting to use these in real programming environments. The following are some of the considerations that we had to address:

- There was some mismatch between the personal process model assumed in PSP and the processes of the participants. For example, some participants did mainly maintenance work and therefore consideration had to be taken of latent defects that were found; participants may have an explicit requirements elicitation and analysis phase; while others performed extensive prototyping and therefore a prototyping phase had to be added. Another example is the participant of Figure 2 who in some cases did only testing of programs. These processes did not match exactly the model assumed in PSP. Therefore, we had to customize (or assist in customizing) the data collection forms for each individual participant.
- After designing new forms, it is important to try them out first with the participants for a period of time in order to make sure that they are satisfied with the new forms. For example, in one case, a participant used the forms happily until he ran out of copies, after which he became frustrated because the form was designed to be on two sides of one sheet and he did not have a two sided photocopy machine in the neighborhood. We therefore had to redesign it to fit on one side only. As another example, one participant confused the seemingly harmless term "Actual" on the Project Plan with the similar word in French "Actuelle" meaning current. He had understood that he had to keep a separate project plan for each day rather than just one for the whole program. It is through trialing of the forms and constant followup that unanticipated difficulties like this are identified and addressed.
- Some participants found it bothersome to fill up the defect recording form for every syntax error (e.g., missed semi-colon or misspelled variable names). We then developed a checklist of common syntax errors that the participants can just tick whenever a syntax error occurs. This substantially reduced the effort to record syntax errors. An example of a form that is used to record the removal of syntax errors injected during coding is given in the appendix. For this form, the engineers put in the cells the phase where the defect was removed. It is assumed that each syntax error takes one minute to fix. If it does take longer, then the engineer could fill up a normal defect recording form.



The above charts provide a signature of this participant's process before the introduction of code reviews. These would serve as a baseline for comparing the effects of future process changes. This participant exhibits most effort variation in her compilation and testing phases compared to coding (as a percentage of total effort). She also consistently spends no time planning. It is interesting to note that for some programs, there is zero coding effort. This is because this participant sometimes only tests code provided by other engineers. Furthermore, it is interesting to note that some programs are not tested by the participant herself. This explains the large variation in testing effort percentage. It is seen that she injects all of the defects during design and coding, and removes all of them during compiling and testing. The majority of her error types are syntax. However, it is the environment errors that take the most time to fix. It would improve overall life cycle time if she injected less environment type errors.

Figure 2: Process signature for one of the participants.

A	Analyze: perform an analysis of the programming problem
D	Design: includes bringing documentation up-to-date
DR	Design Review
CO	Code
CR	Code Review
CM	Compile
TE	Unit Test
IN	Integrate
*B	Break: mid-shift and lunch; these are interruptions that you decide to make rather than being imposed upon you
*CL	Clean Up: desk, papers, computer files and directories
*CF	Configuration Management: builds, library management, file check-in/check-out
*M	Maintenance: of system configuration, computer environment (not of system code)
*ME	Meeting
*MM	Correspondence: email, memos, etc.
*S	Support: when you give advice or help to others related to something you know or have done; for example, users of a library that you have written
*TR	Training: formal lectures, seminars, or reading to learn new skills
*TS	Timesheet: everyone fills one once a week
*X	Interruption

Figure 3: Types of activities used during the detailed time logging. Codes preceded by an asterisk (*) indicate what we have designated as non-programming tasks.

Participant	Design	Code	Compile	Unit Test
1	0.65	1.11	0.62	1.55
2	1.06	0.61	0.43	1.44
3	1.07	0.58	0.25	0.88

Figure 4: Average uninterrupted hours for four programming tasks.

4.2 Training

The training activity started from the first PSP lecture until the end of the study. A number of issues need to be addressed in order to ensure that the participants continue using the PSP concepts in their programming tasks:

- There is a need for constant feedback in order to reinforce what the participants have learned. We did this by having regular meetings with the participants on an individual basis to help interpret the data that they have collected, to track their progress and to answer questions. An example of data that is interpreted in one of these sessions is given in Figure 2. In this example, we wanted to establish a baseline process signature for one of the participants.
- Many participants felt that automated data collection and analysis tools would be useful for them. However, it is difficult to automate data collection in a heterogeneous computing environment. When we attempted automation, we realized that the number of different platforms that were present would be a substantial constraint. Some of the participants implemented tools on their own or used tools that were available in other departments. In the case of line of code measurement, this had to be centralized to ensure that one LOC counter was used (to maintain consistency of measurement).
- The implementation of PSP is difficult unless the managers of the participants are involved in supporting the PSP skills that the participants

have learned in the classroom. Although we did obtain the support of the managers at the start of the study, we did not really involve them in the day-to-day conduct of it. One participant commented that it would be nice if her group leader had come by and asked "how's PSP going?" once in a while.

- Some participants felt that they spend very little time actually doing programming tasks and that interrupts take up much of their time. We therefore requested that they maintain detailed time logs at a 15 minute granularity to determine where their time is spent[2]. For three of the participants who maintained their time logs, we found that approximately 25% of their time is spent on non-programming tasks[3]. The list of task categories that were used is given in Figure 3. After presenting them with these numbers, the participants were more convinced that they in fact do spend most of their time on programming tasks. However, we also found that the amount of contiguous time that these participants spend without being interrupted is very small indeed. The values for the three participants for the design, code, compile, and unit test tasks are shown in Figure 4.

2 Perry et al. [30] report on a study to calibrate self-reported time by software engineers. They found that on average software engineers over-represent the total amount of time that they work by 2.8%. This was calculated by comparing self-reports with reports from direct observation of the programmers at work. Also, they calculated the proportion of time that the self-reports and the observation reports agreed on what the software engineer was actually doing. This was found to vary from 0.95 to 0.58. This gives an indication, based on previous studies, of how accurate the self-reported times in our study are expected to be. It is also interesting to compare these numbers with those obtained by Perry et al. [29]. In that study they found that almost half of the time spent by programmers was on *non-coding* tasks. For the three participants for whom we have collected data, the first spent approximately 69% of his time on tasks other than code and compile, the second spent approximately 82% of his time on tasks other than code and compile, and the third spent approximately 98% of his time on tasks other than code and compile. For these participants, they spend much of their programming time on other tasks. This was mainly unit testing for the former two. The latter participant spent most of the time on design.

3 One participant collected detailed time logs for 47 days, equivalent to approximately 471 hours. The second participant collected detailed time logs for 15 days, equivalent to approximately 107 hours. The third participant collected them for 15 days, equivalent to approximately 118 hours.

4.3 Evaluation

Two evaluations were conducted. The first was of the transfer of training to actual programming tasks. The second was of the benefits of the training.

4.3.1 Evaluating the Transfer of Training

Transfer of training is defined as *"the effective, and continued application to trainees' jobs of the knowledge and skills gained in training"* [7]. In general, it is believed that as little as 10% of expenditures by US industry in training actually result in transfer to the job [1]. This makes the transfer of training an important measure of the effectiveness of PSP training.

We measured the transfer of training by the percentage of participants who were still collecting data on their personal processes. The transfer of training rate for our study was 46.5%. This value was calculated seven months after the commencement of the first PSP lecture. From the perspective of CAE this was considered a success.

This value is comparable to the only study that evaluated the transfer of training, calculated as changes in engineers' habits after the completion of the course [22]. They obtained a value of approximately 45% for personal code reviews and approximately 65% for defect management practices, calculated 5 months later. It is then reasonable to conclude that our rate, derived from an exercise using actual programming tasks and after 7 months, is good. Below we explain how this value was obtained and also the investigations we conducted in order to understand the reasons for the non-transfer of learning that did occur (which was 53.5%).

At the outset, 28 participants were registered to take part in the study. After we started, 5 participants were reassigned to nonprogramming tasks (e.g., updating documentation), 3 were reassigned to field duty, 1 went on maternity leave, and 3 left the company. These twelve participants did not have the opportunity to apply the PSP concepts that were taught. Therefore, 57% (16/28) of the original participants had the opportunity to apply the concepts in their real programming tasks. In terms of the background variables (years of experience and years with the company) there was no statistically significant difference between those who remained and those who left (using a Mann-Whitney U test at the 0.1 alpha level).

After seven months, thirteen participants remained and three had dropped out. This gives an overall

transfer of training rate of 46.5% (13/28). One of the factors that has an impact on the transfer of training is the opportunity to use the learned skills on the job [1]. In this study, it is clear that opportunity to apply the PSP concepts had a large impact on the overall transfer of training. For those who had the opportunity, 81% (13/16) were applying PSP concepts in their real programming tasks seven months afterwards.

At the outset we expected that peer support would be an important determinant of whether participants continued using PSP. We tested this hypothesis for the sixteen participants who had the opportunity to apply the PSP concepts. A point biseral correlation was calculated [4]. This provides a measure of association between a dichotomous and a continuous variable. The dichotomous variable that we used was whether participants were still using the concepts that they were taught after seven months. The continuous variable was the number of people in the same department who were using the concepts that they were taught after seven months. If the association was positive and significant then we can conclude that having other people in the same department using the PSP concepts contributes towards a participant using the PSP concepts. The point biseral correlation was –0.3 and not significant, but it was in the expected direction (i.e., the more people in the department the less likely that they would leave). However, it can be argued that the number of people using PSP concepts is a function of the size of the department and therefore another approach should be used. So we dichotomized the continuous variable into one group of participants where no one in their department was using PSP concepts, and another group where at least one other person was using PSP concepts. We used the Fisher exact test [36] to determine if there was any relationship with use of PSP concepts. For a one tailed test, the result was not significant at an alpha level of 0.1. These results do not support the contention that having other people in the same department using the same personal process concepts contributes towards the transfer of training.

4.3.2 Evaluating the Benefits of PSP Concepts

We evaluated the trends in productivity (measured in LOC of new code per hour) and obtained different results for different engineers. The Daniels test [9] was used to look for monotonic trends[4]. None of the trends were statistically significant, although some were positive (averaging 0.18) indicating an increasing level of productivity over time and others were negative (averaging –0.41) indicating a decreasing level of productivity over time. This general lack of trend, however, is consistent with previous results in [19] where they did not find changes in levels of productivity during all of the PSP exercises.

One measure of the benefits of using PSP concepts that has been used in the past is defect density [19][12]. Using this measure, it is assumed that if defect density goes down, then code quality is improving. We felt that in this context such a measure would not be appropriate because it is difficult to interpret changes in the value of defect density when we only have data from unit testing. If the defect density decreases that could mean that defect detection has deteriorated or that less defects have been injected (e.g., because the problem complexity was very low). Conversely, if defect density increases, this could mean that defect detection has improved or that more defects are being injected.

Marked increases in defect density were witnessed in cases where participants took the code review lecture. Right after the code review lecture, the defect density of participants' programs rose sharply. The average increase in defect density was from approximately 88 defects/KLOC to 265 defects/KLOC after the code review lecture. We found no consistent evidence of large problem complexity, cyclomatic complexity or program size differences between pre and post code review programs. Using interpretations and assumptions from previous studies, this would imply that code quality has actually deteriorated. However, we conjecture that the defect detection capabilities of the engineers have improved after the code review lecture. This result is similar to the slight increase in defect density over time for an industrial PSP course evident in the charts from [31].

Another dependent measure that has been used in previous work on PSP has been the yield [19], which is the percent of defects removed before compilation. The average yield for programs developed without the use of code reviews was 1.65%. However, this includes many programs that were developed with a yield of 0%. If we only use programs developed with a yield greater than 0%, then the average yield without code reviews was approximately 12%. The average yield when code reviews were used was 27.7%. Therefore, the yield

4 We only considered participants that have developed four or more programs during that period.

more than doubles with code reviews. This is not surprising given the way yield is calculated however.

In [23], the authors use the percentage of time spent on test as a measure of the improvements due to PSP. The reasoning is that it indicates an improvement in early defect removal skills. From our data, the average reduction in percentage of time spent on testing was from approximately 37% before code reviews to approximately 17% afterwards. This can be seen as a considerable improvement in early defect removal capabilities.

We also conducted a post-hoc test to investigate the issue of when engineers should review code. In [16], Humphrey notes that engineers question the need to review code before compiling. We can present confirmatory evidence to that given by Humphrey on the relationship between compile defects and test defects. We found a correlation of 0.69 which is significant at the 0.001 alpha level for a one tailed test (after removal of extreme outliers). While this does not imply a causal relationship, it seems that the more defects found in compile the more defects are found in test. This strengthens the argument for reviewing code before compiling it.

4.4 Leveraging

Given that one of the objectives of the PSP pilot study was to create a climate for diffusing measurement concepts to the remainder of the organization, the PSP study, its importance, and its outcomes have to be promoted across the site. One way we did this was through the company newsletter. This also helps give the participants in the study visibility, which is useful in maintaining their enthusiasm.

Further, presentations of the results of the pilot were given to senior management. These presentations generated sufficient enthusiasm that followup studies are planned.

5 Lessons Learned

Some of the main lessons that we have learned in this study are summarized below:

- The need for customization of personal processes was recognized in [15]. It is also important to customize the PSP data collection forms to the personal processes of individual participants, and also to the work environment. This point was also noted in [31]. A structured process for eliciting process information has been presented in [25], which can be used to elicit personal processes.
- All forms must be piloted with the participants in their real work environment. Even if the forms were designed to fit their personal processes, actual use in realistic programming tasks may reveal deficiencies in the design of the forms.
- It is important to have automated tools that support the participants' data collection and also data analysis. The paper intensive nature of PSP was also identified as a problem for professional engineers in [31]. Furthermore, it would be preferable if the data collection forms are available in editable format for the participants so that they can customize the forms themselves as they gain a better understanding of their processes (e.g., to remove or add activities).
- It would be most preferable to give the supervisors or managers of the participants at least a formal short overview of PSP so that they understand it and see its benefits. This would help gain stronger commitment from management for PSP. This is similar to the top-down approach to introducing PSP in organizations suggested in [17], and practiced in one organization [26].
- Lectures should cover all of the typical life cycles that are in effect in the organization, not only the one presented in the PSP manuscript. This makes the classroom teaching more relevant to the participants' real programming tasks.
- Feedback sessions to the participants are important in order to reenforce the concepts that they have learned. Also, they have to see the data that they collect being used, otherwise they may lose interest.

We have identified above a number of lessons that we believe are important in a PSP implementation. Furthermore, we have, where possible, corroborated these lessons with findings from other recent reports on teaching PSP.

6 Conclusions

Professionals and students taking courses on the Personal Software Process have demonstrated impressive improvements in their personal capabilities. There has been little systematic study, however, of the implementation of Personal

Level 1 - Evaluating Reaction
At this level of evaluation, one measures the reaction of participants to the PSP training that they have received. It is essentially a measure of customer satisfaction. Usually, one would administer a questionnaire at the end of the training course.

Level 2 - Evaluating Learning
At this level, one measures the extent to which participants change their attitudes, improve knowledge, and/or increase their skill levels as a consequence of taking the PSP training (assuming that they have a positive reaction - otherwise there is little chance of learning). This type of evaluation would usually be done during the training phase. An easily implemented approach for doing this kind of evaluation is to compare participant performance at the end of the course with performance at the beginning of the course.

Level 3 - Evaluating Behavior
At this level, one evaluates the extent to which change in behavior has occurred due to taking the PSP course. It is assumed that students have learned, otherwise they are not likely to change their behavior. The evaluation of behavior can be achieved by calculating the proportion of participants who had the opportunity to apply the PSP concepts and how many actually apply them in their real programming tasks. This type of evaluation is done during the evaluation phase of the implementation.

Level 4 - Evaluating Results
At the fourth and final level, one would evaluate the benefits gained by the participants in their real programming tasks (assuming that they have changed their behavior). In the case of PSP, results can be evaluated at at least two units of analysis: the personal and project units. For instance, for a personal unit of analysis, one can evaluate whether each participants' defect removal skills have improved compared to before applying PSP concepts. For a project unit of analysis, one can evaluate whether field defects for a product were reduced after the team applied the PSP concepts in the project.

Figure 5: A four-level framework for evaluating PSP implementation in industrial settings.

Software Process concepts in real programming environments. In this paper we presented the details of a pilot implementation of some of these concepts (in our study these were measurement and structured code reviews) in a commercial organization. We found that seven months after starting 46.5% of the participants were still using PSP concepts in their real programming tasks. Furthermore, those who used code reviews demonstrated substantial improvements in their defect removal skills.

The objectives of the pilot study were also met: (a) the chosen PSP concepts were tailored to the organization, (b) there has been an increased awareness of measurement within the organization, (c) we evaluated how many participants continue applying the PSP concepts in their work using our implementation approach, and (d) we evaluated the benefits of the implemented PSP concepts.

We have also identified a number of factors that should be considered in order to ensure a successful implementation in an industrial setting. Of these, perhaps the most important are: customizing the course materials to the organization and the personal practices of the engineers, providing automated tools for use by the engineers, and obtaining management commitment and support for implementation.

While our results are specific to one organization, they do provide some initial guidelines for others embarking on an implementation of the Personal Software Process in industry. Furthermore, it would be informative to see if future implementation studies obtain similar results of Personal Software Process effectiveness to the ones obtained here.

To promote more systematic study of the implementation of PSP in industrial settings and to facilitate their comparison, we have adapted Kirkpatrick's framework for evaluating training programs [20]. His framework goes from the very simple evaluation (level 1) to the most informative and sophisticated (level 4). This framework is summarized in Figure 5. In the current paper, we have reported

evaluations at level 3 and level 4 (at the personal unit of analysis).

Acknowledgements

The authors wish to thank John Daly, Jerome Pesant, and Pascale Tardif for their constructive comments on an earlier version of this paper.

References

[1] T. Baldwin and J. Ford: "Transfer of Training: A Review and Directions for Future Research". In *Personnel Psychology*, 41:63-105, 1988.

[2] A. Cherns: "Social Research and its Diffusion". In *Human Relations*, 22(3):209-218, 1969.

[3] P. Clark: Action Research and Organizational Change, Harper & Row, 1972.

[4] G. Ferguson and Y. Takane: *Statistical Analysis in Psychology and Education*, McGraw-Hill, 1989.

[5] R. Galliers: "In Search of a Paradigm for Information Systems Research". In *Research Methods in Information Systems*, E. Mumford et al. (eds.), Elsevier Science Publishers, 1985.

[6] R. Galliers and F. Land: "Choosing Appropriate Information Systems Research Methodologies". In *Communications of the ACM*, 30(11):900-902, November 1987.

[7] P. Garavaglia: "How to Ensure Transfer of Training". In *Training and Development*, pages 63-68, October 1993.

[8] D. Georgenson: "The Problem of Transfer Calls for Partnership". In *Training and Development Journal*, pages 75-78, October 1982.

[9] J. Gibbons: *Nonparametric Statistics*, Sage Publications, 1993.

[10] B. Hall: "Participatory Research: An Approach for Change". In *Convergence: An International Journal for Adult Education*, 8(2):24-31, 1975.

[11] M. Hult and S. Lennung: "Towards a Definition of Action Research: A Note and Bibliography". In *The Journal of Management Studies*, 17:241-250, 1980.

[12] W. Humphrey: "The Personal Process in Software Engineering". In *Proceedings of the 3rd International Conference on the Software Process*, pages 69-77, 1994.

[13] W. Humphrey: "The Personal Software Process". In *Software Process Newsletter*, IEEE TCSE, No. 1, pages 1-3, September 1994.

[14] W. Humphrey: *A Discipline for Software Engineering*, Addison Wesley, 1995.

[15] W. Humphrey: "Introducing the Personal Software Process". In *Annals of Software Engineering*, 1:311-325, 1995.

[16] W. Humphrey: "The Power of Personal Data". In *Software Process Improvement and Practice Journal*, 1:69-81, 1995.

[17] W. Humphrey: "Using a Defined and Measured Personal Software Process". In *IEEE Software*, pages 77-88, May 1996.

[18] A. Jenkins: "Research Methodologies and MIS Research". In *Research Methods in Information Systems*, E. Mumford et al. (eds.), Elsevier Science Publishers, 1985.

[19] S. Khajenoori and I. Hirmanpour: "An Experiential Report on the Implications of Personal Software Process for Software Quality Improvement". In *Proceedings of the Fifth International Conference on Software Quality*, pages 303-312, October 1995.

[20] D. Kirkpatrick: *Evaluating Training Programs: The Four Levels*. Published by Berrett-Koehler, 1994.

[21] D. Leonard-Barton and W. Kraus: "Implementing New Technology". In *Harvard Business Review*, pages 102-110, November/December 1985.

[22] S. Macke: "Personal Software Process at Motorola PPG". In *Proceedings of the Software Engineering Process Group Conference*, 1996.

[23] S. Macke, S. Khajenoori, J. New, I. Hirmanpour, J. Coxon, A. Ceberio, and B. Manente: "An Industry/Academic Partnership that Worked: An In Progress Report". In *Proceedings of the 9th Conference on Software Engineering Education*, April 1996.

[24] S. Macke, S. Khajenoori, J. New, I. Hirmanpour, J. Coxon, and R. Rockwell: "Personal Software Process at Motorola Paging Products Group". In *Proceedings of the Software Engineering Process Group Conference*, 1996.

[25] N. Madhavji, D. Hoeltje, W-K Hong, and T. Bruckhaus: "Elicit: A Method for Eliciting Process Models". In *Proceedings of the 3rd International Conference on the Software Process*, pages 111-122, 1994.

[26] A. Matvya: "Industrial Strength PSP at Union Switch & Signal". In *Proceedings of the Software Engineering Process Group Conference*, 1996.

[27] D. Michalak: "The Neglected Half of Training". In *Training and Development Journal*, pages 22-28, May 1981.

[28] J. Mosel: "Why Training Programs Fail to Carry Over". In *Personnel*, 34(3):56-64, November-December 1957.

[29] D. Perry, N. Staudenmayer, and L. Votta: "People, Organizations, and Process Improvement". In *IEEE Software*, pages 36-45, July 1994.

[30] D. Perry, N. Staudenmayer, and L. Votta: "Understanding and Improving Time Usage in Software Development". In *Software Process*, A. Fuggetta and A. Wolf (eds.), Wiley, 1996.

[31] M. Ramsey: "Experiences Teaching the Personal Software Process in Academia and Industry". In *Proceedings of the Software Engineering Process Group Conference*, 1996.

[32] R. Rapoport: "Three Dilemmas in Action Research". In *Human Relations*, 23(6):499-513, 1970.

[33] D. Roy: "The Personal Software Process: An 'Ego-Centered' Improvement Paradigm". In *Proceedings*

of the Software Engineering Process Group Conference, 1996.

[34] K. Sherdil: "Personal 'Progress Functions' in the Software Process". Master's Thesis, School of Computer Science, McGill University, 1994.

[35] K. Sherdil and N. H. Madhavji: "Human-Oriented Improvement in the Software Process". In *Proceedings of the 5th European Workshop on Software Process Technology*, Springer Verlag, 1996.

[36] S. Siegel and J. Castellan: *Nonparametric Statistics for the Behavioral Sciences*, McGraw Hill, 1988.

[37] R. Van Horn: "Empirical Studies on Management Information Systems". In *Data Base*, 5:172-180, 1973.

Appendix: Example Defect Recording Form for Syntax Errors

DEFECT RECORDING FORM

Name:_____ Module ID:_____ Sheet___ of____

COMMON ERRORS

missing {or}	
missing (or)	
missing "or'	
missing ;	
missing /* or */	
missing include	
undeclared variable	
misspelled variable	
incorrect = or ==	

REMOVAL PHASE:

P : Planning
DR : Design review
CR : Code review
T : Test
D : Design
CO : Code
CM : Compile