To appear (as of March 30, 2000) in IEEE Transactions on Software Engineering

An Experiment Measuring the Effects of Personal Software Process (PSP) Training

Lutz Prechelt (prechelt@computer.org)
Barbara Unger (unger@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-3934, Fax: +49/721/608-7343

March 30, 2000

Abstract

The Personal Software Process is a process improvement methodology aiming at individual software engineers. It claims to improve software quality (in particular defect content), effort estimation capability, and process adaptation and improvement capabilities. We have tested some of these claims in an experiment comparing the performance of participants who had just previously received a PSP course to a different group of participants who had received other technical training instead. Each participant of both groups performed the same task.
We found the following positive effects: The PSP group estimated their productivity (though not their effort) more accurately, made fewer trivial mistakes, and their programs performed more careful error-checking; further, the performance variability was smaller in the PSP group in various respects. However, the improvements are smaller than the PSP proponents usually assume, possibly due to the low actual usage of PSP techniques in the PSP group.
We conjecture that PSP training alone does not automatically realize the PSP's potential benefits (as seen in some industrial PSP success stories) when programmers are left alone with motivating themselves to actually use the PSP techniques.

Keywords: process improvement, quality management, effort estimation, reliability, productivity, experiment

1 The Personal Software Process (PSP) methodology

The Personal Software Process (PSP) methodology for improving the software process was introduced in 1995 by Watts Humphrey [6]. PSP is an application of the principles of the Capability Maturity Model (CMM, [5]) on the level of an individual software engineer. In contrast to the CMM, however, which allows only for assessment of process quality, the PSP makes concrete methodological and learning suggestions, down to the level of a 15-week course with rather specific procedural content. The goals of the PSP are that an individual software engineer learns

- how to accurately estimate, plan, track, and re-plan the time required for individual software development efforts,
- how to work according to a well-defined process,
- how to define and refine the process,
- how to use reviews effectively and efficiently for improving software quality and productivity (by finding defects early),

- how to avoid defects,
- how to analyze measurement data for improving estimation, defect removal, and defect prevention,
- how to identify and tackle other kinds of process deficiencies.

The main basic techniques used are gathering objective measurement data on many aspects of the process (for obtaining a solid basis for process change decisions), spelling out the process into forms and scripts (for precise control of such changes and for making the process repeatable), and analyzing the collected data (for deciding where and how to change the process).

1.1 Previous evidence for PSP effectiveness

The PSP methodology relies a lot on objective measurement data and so does the argumentation of the PSP proponents. Watts Humphrey has published data from several PSP courses in 1996 [7] and several experience reports from other PSP practitioners report similar results. Since no other information was available, these results rely on the very data collected by the course participants during their course exercises 1 through 10, regarding real and estimated development time as well as inserted and removed defects, both for each of the various development phases. These data show for instance that the estimation accuracy increases considerably, the number of defects introduced per 1000 lines of code (KLOC) decreases by a factor of two, the number of defects per KLOC to be found late during development (i.e., in test) decreases by a factor of three or more, and productivity is not reduced despite the substantial overhead factor for bookkeeping involved when the tasks are as small as they are. See [4] for a detailed analysis of these effects based on data from 23 courses.

Additional evidence comes from industrial success reports on PSP usage, in which the expected PSP benefits were actually found in several small industrial projects whose engineers used PSP [3].

Unfortunately, both kinds of evidence have severe drawbacks. Observations directly from the course are distorted in several ways. First, the process definition underlying the data changes from exercise to exercise, making exact interpretation difficult. Second, the PSP course is the Hawthorne effect [9] at its best: the participants constantly monitor their performance and their central objective is improving this performance — in contrast to industrial software practice, where individual performance is only a means to an end. Third, individual participants may consciously or subconsciously manipulate their time measurement and defect recording towards better results. Fourth, the exercises 4 through 10 are not only quite simple in terms of design and implementation complexity, but also have unusually clear and easy-to-understand requirements, which may make the results overly optimistic. Finally, and most importantly, there is no control. Nobody knows how performance would change during these 10 programming assignments if no PSP training and self-monitoring occurs at all. A validation of the PSP by direct comparison to non-PSP-trained persons is missing.

The industrial case studies are difficult to interpret (due to their complex context) and also lack control. Either comparisons to non-PSP data are missing or they are based on a before/after comparison of the same persons (not controlling for maturation) or an A/B comparison to different projects. Besides many other relevant differences, such comparison projects might have engineers that are less capable than the vanguard that chose to learn PSP first. Again, a study with a higher degree of control is called for.

1.2 Experiment motivation and overview

When we learned the PSP ourselves and then started teaching it to our graduate students, we soon obtained the impression that the methodology has a lot of benefits, but is not without problems, too. In particular, many programmers appear to be unable to keep up the discipline required for the data gathering and for following a spelled-out process script; this may also underlie some of the PSP data quality problems observed by Johnson and Disney [8]. Furthermore, many course participants misunderstand the concrete techniques and procedures taught in the PSP course as dogmas, although they are meant only as starting points for one's own process development. They do not understand that (and how) they should adapt the proposed techniques to their own preferences and needs.

We estimated that a majority of our course participants would probably use little or nothing of the PSP techniques in their later daily work and we wondered whether, under these circumstances, PSP training would be more effective than any other technical training with respect to the PSP goals.

We hence conducted the experiment as described in Section 2, where we compared a group of students right after a PSP course to a group of other students right after a different (technical rather than methodological) course; see Section 2.2. Each participant had to solve the same programming task (see Section 2.3), which was quite different from all of the assignments of either course.

We observed several aspects of each subject's development process and software product to analyze the effects of the PSP course in comparison to a more technical course, in particular the accuracy of the subjects' effort estimation and the reliability of the programs they produced. The results are discussed in Section 3.

The experiment and its results are described in more detail in a technical report [11], which also includes the actual experiment materials. The report also contains various less important additional measurement results not discussed in this article. The materials and raw result data are also available in electronic form from http://wwwipd.ira.uka.de/EIR/.

2 Description of the experiment

2.1 Experiment design

The experiment uses a single-factor, posttest-only, inter-subject design [1]; see Table 1 for an overview. The independent variable was whether the experimental subjects had just previously participated in a PSP course (experiment group, subsequently called "P") or in an alternative course (comparison group, subsequently called "N" for "non-PSP"). Each subject of either group solved the same task and worked under the same conditions. The assignment of subjects to groups could not be randomized; this threat is discussed in Section 2.6. The observed dependent variables for each subject were a variety of measures of personal experience, various estimations of development time for the task (in particular estimated total time), various measurements of the development process (in particular total time), and various measurements of the delivered product (in particular program reliability).

	group N	group P
treatment	PSP course	KOJAK/other
task	phoneword	phoneword
observed	work time, estim. time, reliability, etc.	

Table 1: The experiment design. For details see the following subsections.

2.2 Subjects

Overall, 48 persons participated in the experiment, 29 in the PSP group P and 19 in the non-PSP group N. All of them were male Computer Science master students. The P group had previously participated in a 15-week graduate lab course introducing the PSP methodology, involving ten small programming assignments and five process improvement assignments, both as described in Humphreys book [6, Appendix D]. All but eight members of the

N group had participated in a 6-week graduate compact lab course on component software in Java (KOJAK), involving five larger programming assignments. This course covered technical topics such as Swing, Beans, and RMI. It was shorter than the PSP course, but much more intensive, so that the overall amount of practical programming experience gained was similar. The other eight of the N participants came from other lab courses of similar magnitude. In contrast to all others, who were obliged to participate (but not to succeed) in the experiment for passing their course, these eight were volunteers. The subjects participated in the experiment between 0 to 3 months after their respective course was finished. There were two instances of the PSP course (1997 and 1998), both run by the same teacher and with the same content.

On average, these 50 students were in their 8th semester at the university, they had a median programming experience of 8 years total and estimated they had a median of 600 hours of programming practice *beyond* their assignments from the university education and had written a median of 20.000 LOC total. None of these measures was significantly different between the two groups. During the experiment, 24 of the participants used Java (JDK), 13 used C++(g++), 9 used C (gcc), 1 used Modula-2 (mocka), and 1 used Sather-K (sak).

8 participants dropped out of the experiment and will be ignored in the subsequent data analysis. They chose to give up after zero to three unsuccessful attempts at passing the acceptance test that forms the end of the experiment (see Section 2.4). All dropouts said they were frustrated by the difficulty of the task; we could not find any connection to group membership. The fraction of dropouts is the same in the P group (5 out of 29, 17 percent) as in the N group (3 out of 19, 16 percent) so that ignoring the dropouts does not introduce a bias into the experiment. This leaves 40 participants for the evaluation.

2.3 Experiment task

The task to be solved in this experiment is called *phoneword*. It consists of writing a program that encodes the digits of long telephone numbers (program input) into corresponding sequences of words (program output, the words come from a large dictionary also provided as input) according to the fixed, given letter-to-digit mapping shown in Table 2: A single digit is allowed to encode itself in the output iff no other digit precedes it and no word from the dictionary can represent the digits starting from that point. All possible complete encodings must be found and printed. Many phone numbers have no complete encoding at all, even with a large dictionary. Dashes and quotes in the words as well as dashes and slashes in the phone numbers must be ignored for the encoding but still be printed in the result. Any phone number and any word in the dictionary was known to be at most 50 characters long. The dictionary was known to be at most 75000 words long. Dictionary and phone numbers are read from two text files containing one word or phone number per line.

Here is an example program output for the input "3586-75", using a German dictionary:

```
3586-75: Dali um
3586-75: Sao 6 um
3586-75: da Pik 5
```

The requirements for this program were described thoroughly in natural language and remaining ambiguities were resolved by examples of correct and incorrect encodings.

The requirements description stated program reliability as the single top objective for the participants. Productivity, program efficiency, etc. were to be less important. For complete text of the requirements specification see [11, pp. 66–68].

2.4 Experimental procedure

The experiment was run between February 1997 and October 1998, mostly during the semester breaks. Most subjects started at

E	J N Q	R W X	D S Y	F T	A M	C I V	B K U	L O P	G H Z
0	1	2	3	4	5	6	7	8	9

Table 2: Prescribed letter-to-digit mapping for the phoneword task. The mapping was constructed such as to balance the letter frequency for each digit across the German dictionary used.

about 9:30 in the morning. Both groups were handled exactly alike. In particular, the PSP subjects were not specifically asked to use PSP techniques. The experiment materials were printed on paper and consisted of two parts. Part one was issued at the start of the experiment and contained a personal information questionnaire, the task description, and an effort estimation questionnaire. After filling these questionnaires in and reading the task description, the subjects worked on the task using a specific Unix account that provided the automatic monitoring infrastructure, which nonintrusively protocoled login/logout times, all compiled source versions with timestamps, etc. The subjects could modify the setup of the account as necessary, make work pauses whenever required, and use any methods and approaches for solving the problem they deemed appropriate. In particular, a few subjects imported source code of reusable procedures (for file handling etc.) from other accounts and a few subjects imported and installed small personal tools. The input, output, and dictionary data used in the example in the task requirements description was provided to the subjects, the large 73113-word dictionary later used for evaluating all programs was also available.

When a subject thought his program worked correctly, he could call for an acceptance test. This was based on randomly generating a set N of 500 phone numbers, computing the multiset of corresponding outputs $S(N)$ using the subject's program, and computing the correct outputs $C(N)$ using a reference implementation ("gold" program). The gold program had been written by the authors using stepwise refinement with semi-formal verification based on preconditions and postconditions. The gold program had run correctly right from its first test and no defect was ever found in it (despite harsh protests from several participants and thorough investigations of their validity). The entire expected output $C(N)$ and actual output $S(N)$ was shown, but the acceptance test used a 20946-word dictionary not available to the subjects. The output reliability r was defined as the fraction of correct outputs within all actual outputs (whether expected or incorrect), i.e. $r = |S(N) \cap C(N)|/|S(N) \cup C(N)|$. To pass the acceptance test, r had to be at least 95 percent.

Misuse of the acceptance test as a convenient automatic testing facility was avoided by the following reward scheme: Each participant received a payment of DM 50 (approx. 30 US dollars) for successful participation, i.e., passing the acceptance test, but for each failed acceptance test, DM 10 were deducted.

12 of the successful subjects finished the day they started, 10 others required a second day, and the other 18 took between three and eleven days. Similarly, 15 subjects passed the first acceptance test, 24 the second to fifth, only 1 required six. Both of these measures showed no significant differences between the two groups.

After passing the acceptance test, the subject were given part 2 of the experiment materials, a short postmortem questionnaire. After filling that in, the subjects were paid and their participation was complete.

2.5 Hypotheses

The experiment investigated the following hypotheses (plus a few less important ones not discussed here, see [11]).

- **Reliability:** PSP-trained programmers produce a more reliable program for the phoneword task than non-PSP-trained programmers.
- **Estimation accuracy:** PSP-trained programmers estimate the time they need

for solving the phoneword task more accurately than non-PSP-trained programmers.

We also sort of expect that PSP-trained programmers may solve the phoneword task faster than non-PSP-trained programmers. Productivity improvement is not an explicit claim or goal of the PSP, but for the given task it might be a side effect of improved quality, because locating a problem detected in the acceptance test is relatively difficult.

2.6 Threats to internal validity

The control of the independent variable is threatened by the fact that group assignment was not done by randomization but rather was due to earlier self-selection of the course taken. In principle, there might be systematic differences between the types of programmers taking either decision. However, we do not believe that such differences, if any, are substantial. For instance, several of the participants of the KOJAK course and several of the volunteers from other courses later chose the PSP course and vice versa.

The choice of programming language might also influence our results, because the different languages have different frequency in the two groups. However, the structure of our task is such that the language used has only modest impact. In particular, neither object-oriented language features nor particular memory management mechanisms are very relevant for the given task. (See, however, the discussion of program crashes at the end of Section 3.1.)

2.7 Threats to external validity

There are several important threats to the external validity (generalizability) of our experiment.

First, and most importantly, different work conditions than found in the experiment may positively or negatively influence the effectiveness of the PSP training. This is discussed in Section 4.

Second, the PSP education of our subjects was only a short time ago. Long-term effects would be more interesting to see.

Third, our task was unusual in several respects (small size, precise requirements, acceptance test indicates expected outputs). It is unknown how these properties might influence the comparison.

Finally, professional software engineers may have different levels of skill than our participants. A higher skill and experience level may leave less room for improvement, but may also sharpen the eye as to where improvements are most desirable or most easy to achieve with PSP techniques. Conversely, lower skill (which will occur, because our students are more skilled than most of the non-computer-scientists that frequently start working as programmers today) may leave more room for improvement but may also impede applying PSP techniques correctly or at all.

3 Results and discussion

We will now describe and discuss the results for estimation accuracy, reliability, and productivity with respect to the hypotheses.

The data will be presented using boxplots (see the figures below) indicating the individual data points, the 10% and 90% quantiles[1] (as whiskers), the 25% and 75% quantiles (by the edges of the box), the median (50% quantile, by a fat dot), the mean (by a capital M), and one standard error of the mean (by a dashed line). The two distributions of the groups N and P are shown side-by-side.

Formal tests of the hypotheses are performed by one-sided statistical hypothesis tests for differences of the mean or the median. We use a Wilcoxon rank sum test for comparing medians and a bootstrap resampling test [2] for comparing the means without relying on the assumption of a normal distribution. When we report for instance "mean test $p = 0.07$" this means

[1]For instance, the 10% quantile of a set of values is an interpolated x such that 10% of the values are smaller or equal to x and 90% are larger or equal to x.

that the test comparing the means of the two groups indicates that the observed difference has a 7% probability p of being purely accidental (i.e., no real difference exists). At values at or below 5% we will call such p-values "significant" and believe that the differences are real. See [11, pp.20–25] for a more detailed description of boxplots and the tests.

3.1 Reliability

We measured the reliability (as defined in Section 2.4) of the delivered programs on eight different randomly generated input data sets, in which each of the possible letters /, –, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 had the same probability at each position of each number in each file (with one exception mentioned below). The data sets contained either 100, 1000, 10000, or 100000 telephone numbers. Since some of the programs were extremely slow (over 15 seconds per input), we could not exercise the largest tests on all of the programs and will hence report the results for the test sets of size 1000 only (but see [11, pp.31–39] for the other results).

There were two different 1000-number data sets, a standard one and a surprising one. The surprising one used exactly the distribution of telephone numbers mentioned above with uniformly random lengths between 1 and 50 characters. This set is surprising because when thinking of these inputs as telephone numbers, intuitively one would not expect to see something not containing any digit at all, such as "/" or "–/-/", even though the requirements description of the task had defined "A telephone number is an *arbitrary(!)* string of dashes, slashes, and digits." (emphasis is in the original). An empty encoding must be output for a phone number without digits. In contrast, the "standard" input set suppressed numbers not containing any digit and generated a new one until at least one digit was present.

Note that both of these tests are harder than the acceptance tests: The dictionary is larger (73113 words versus 20946 words), there are twice as many phone numbers, and the critical phone numbers of length 1 or 50 had artificially been made less frequent in the acceptance test.

The results for the standard data set are shown in the rather degenerated box plots of Figure 1. As we see, the reliability of the programs is generally high in both groups, both medians are at 100%. We do not find any evidence for superior reliability in the P group. Instead, there even is a slight advantage for the N group, but the difference is not significant (median test $p = 0.32$, mean test $p = 0.33$).

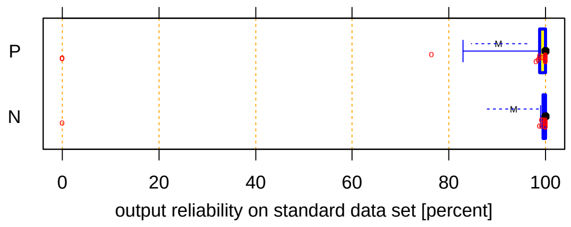

Figure 1: Output reliability (as defined in Section 2.4) for the data set with at least one digit per phone number.

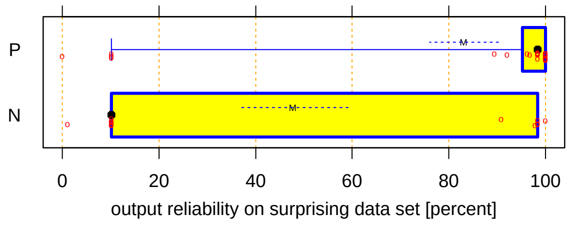

Figure 2: Output reliability for the data set with possibly no digit in a phone number.

The results for the surprising data set are different (Figure 2). Again, some programs work perfectly, but many other programs from the N group and also a few from the P group crash[2] at the first telephone number that contained no digit, which resulted in a reliability of 10.7% for the given data set. As a result the reliability is significantly lower in the N group (median test $p = 0.00$, mean test $p = 0.01$).

Note that faulty Java programs were more likely to crash than faulty C or C++ programs, due to the Java run time checks of array indices etc. Since the fraction of Java programs

[2]Our measurement ignores the crash and just records its consequences: no further output arrives.

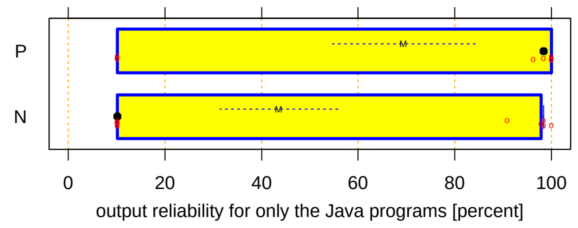

Figure 3: Output reliability for the data set with possibly no digit in a phone number, measured only for the Java programs.

is higher in the N group, the above result may thus be biased in favor of the P group. Therefore, we also compared the Java programs alone (see Figure 3) and found that the above group difference indeed becomes less significant, but does not disappear (median test $p = 0.04$, mean test $p = 0.07$). For the other languages there are not enough data points for a meaningful comparison.

These results provide some evidence for higher reliability in the P group. Specifically, although P group program reliability is not generally higher, the P group programs perform better error checking and handling of unexpected situations. See also the discussion of program length in Section 3.3. We conclude that the reliability hypothesis is supported, though not clearly confirmed by the experiment.

3.2 Estimation accuracy

For comparing the effort estimation capabilities of the groups, we consider the estimation of the total working time the subjects made after reading the task description. We compute the mis-estimation in percent from the quotient of actual and estimated time, see Figure 4.

The median mis-estimations are essentially identical (median test $p = 0.48$). The worst estimations of the N group are worse than in the P group, so that the mean mis-estimation tends to be larger in the N group; but the difference is not significant (mean test $p = 0.18$). This is no convincing evidence that the P group produces better estimates.

PSP estimation is based on program size estimation and historical data on personal productivity (measured in lines of code produced per hour). If we compare the subjects' expected productivity to the actual productivity (Figure 5), we find the estimations of the N group are clearly worse than in the P group (median test $p = 0.00$, mean test $p = 0.00$). This shows that the PSP group knows their historical data, but did not produce a size estimation that was good enough for converting this knowledge into an estimation advantage. We conclude that, overall, the hypothesis of better time estimation in the PSP group is not supported by the experiment.

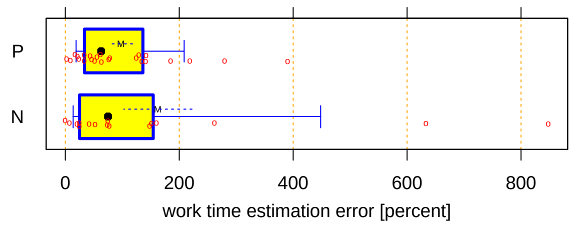

Figure 4: $\left|\frac{t_{work}}{t_{estim}} - 1\right|$: Amount of mis-estimation of the total working time for the task. Most estimations were too optimistic, only four in each group were too pessimistic.

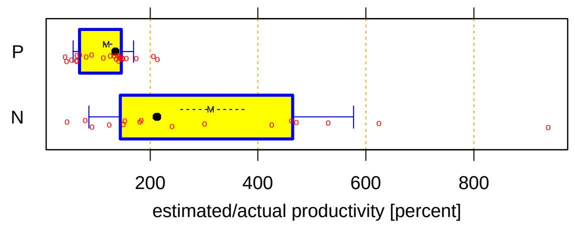

Figure 5: $\frac{prod_{estim}}{prod_{actual}}$: Quotient of estimated and actual productivity, each measured in lines of code per hour.

3.3 Productivity

Since all subjects worked on exactly the same task, we might take the view that their productivity should best be expressed simply as the "number of tasks solved per time unit", which is just the inverse of the total working time and is shown in Figure 6.

From this point of view, in contrast to our expectations the productivity is not larger, but

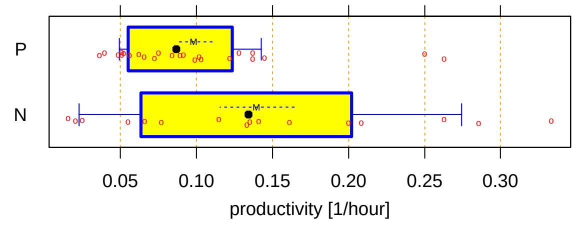

Figure 6: $\frac{1}{t_{work}}$: Productivity measured as the inverse of total working time (number of programs per hour, one could say).

rather tends to be smaller in the P group (median test $p = 0.10$, mean test $p = 0.06$). We do not know the degree to which the PSP bookkeeping overhead accounts for this tendency, but probably the degree is low, because few subjects actually had any significant PSP overhead (see Section 3.5).

However, the P group generally wrote longer programs, partly because of more careful error checking. For instance, in the Java programs the average number of 'catch' statements with non-empty exception handlers is significantly larger in the P group than in the N group (4.4 versus 2.5, $p = 0.02$). As we saw above, this additional effort tended to result in better reliability. So maybe the more conventional view of productivity as the number of lines of code written per hour is more appropriate. This quotient is shown in Figure 7; the productivity difference has essentially disappeared (median test $p = 0.42$, mean test $p = 0.30$).

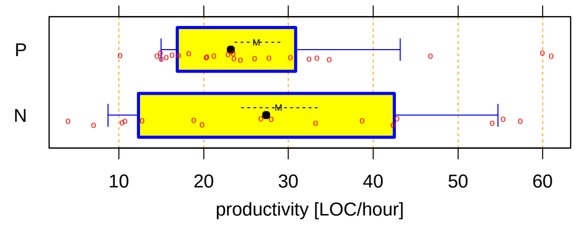

Figure 7: $\frac{LOC}{t_{work}}$: Productivity measured as the number of statement LOC written per hour.

Curiously, if we consider the working time directly (instead of its inverse), as shown in Figure 8, the N group has a lower median (median test $p = 0.10$), but a slightly higher mean due to some very slow subjects (mean test $p = 0.28$).

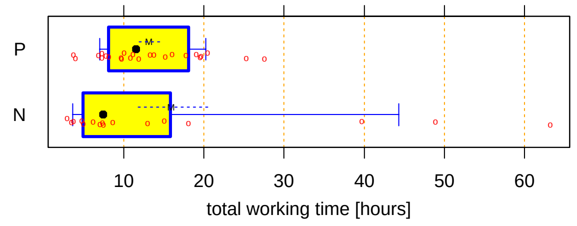

Figure 8: t_{work}: Total number of working hours.

As we see, the notion of productivity is slippery for software and must be handled with care. However, the hope that the productivity of PSP-trained persons would be higher is not supported in the experiment (and is also not claimed by the PSP in general).

3.4 Variance within each group

One rather unexpected insight from this experiment was that even where no improvement of the average occurred, the variability was usually smaller in the P group than in the N group. The effect occurred for most of the measures we have investigated. We can quantify this by providing a bootstrap resampling test for differences of the length of the box ("interquartile range", iqr test), which is a rather robust measure of variability.

For instance for the productivity and estimation comparisons shown in the figures above, smaller P group variability is visible as a tendency for the reliability on surprising inputs (Figure 2, iqr test $p = 0.12$) work time misestimation (Figure 4, iqr test $p = 0.24$) and the total work time (Figure 8, iqr test $p = 0.39$) and is statistically significant for the productivity estimation (Figure 5, iqr test $p = 0.01$), the productivity in terms of working time (Figure 6, iqr test $p = 0.05$), and the productivity in LOC per hour (Figure 7, iqr test $p = 0.04$).

The reduction of variability in the PSP group may be a benefit. Teams with lower interpersonal performance variability can be more flexible, because any member could take over a task without changing the schedule. Schedule risk may also be an issue, as is outlined in [12].

3.5 PSP usage

For only 6 of the 24 PSP participants (25 percent), we found evidence[3] that they had actually used PSP techniques. This low percentage may imply that the size of the differences found above reflect the *degree* of actual PSP use more than the *effect* of PSP use.

Furthermore, we found a surprising correlation. Among those 5 PSP participants who had given up in the experiment, as many as 4 (or 80 percent) showed evidence of actual usage of PSP techniques — a significantly higher fraction than among the successful participants (Fisher exact $p = 0.036$). Apparently the least capable subjects had a much higher inclination to use PSP techniques, presumably because they feel more clearly that these techniques help them.

3.6 Other results

We have collected other measures (not directly connected to our hypotheses) as well and found several areas where some advantage was visible for the P group. For instance they estimated the average time required for fixing a defect or the reliability of their program more accurately than the N group. Furthermore, they made fewer trivial mistakes that led to compilation errors and tended to write more comments into their programs. See [11] for details.

In the informal postmortem interview, many of the P subjects (but none of the N subjects) said something along the lines of "I *really* should have performed design and code reviews. Damn that I didn't."

4 Conclusion

Our experiment comparing a group of PSP-trained programmers (P group) to a similar group of programmers who received other training (N group) produced the following major findings:

[3]In all cases this evidence included a time and defect log, in some cases also a PSP estimation form.

- The programs produced by members of the P group are slightly more reliable than those of the N group as far as robustness against unusual (but legal) inputs is concerned. For more standard types of inputs we did not find a reliability difference.
- The members of the P group estimated their productivity (in lines of code per hour) better than the N group, but did not produce better total effort estimates.
- The total time for finishing the task tended to be longer in the P group than in the N group. However, at least to some degree this additional time is invested in the error checking that leads to the improved robustness. The productivity in lines of code per hour was hardly different in both groups.
- For many performance metrics the variability within the P group was substantially smaller than the variability within the N group.
- Apparently a majority of the P group participants did not use PSP techniques at all.

When one compares these results to some of the improvements known from the PSP course, one may be disappointed; for instance, participants of a PSP course on average achieve an at least fourfold reduction of the number of defects to be found in test during their course exercises 1 through 10. There were no such dramatic differences in this experiment. We see two major reasons. First, the PSP course with its constant measurement and feedback is a typical Hawthorne effect situation [9], so that results from the course overestimate the improvements available in the long run. Second, too few of our subjects from the PSP group actually used PSP techniques during the experiment; most of them did not keep up the necessary self-discipline.

We offer three explanations for the low degree of actual PSP usage: First, it may be a result of different temperaments of the programmers. In our experience, a few pick up and use the PSP techniques quite easily and enthusiastically, a majority can adapt them only with effort and will later use them to a modest degree at best, and some appear to be completely unable to maintain the discipline required for applying the PSP techniques. The proportions may be culture-dependent, so subjects from other countries may turn out different in this respect. Our course has won several awards for best teaching (as evaluated by the students) in our department, so we rule out low quality of teaching as a reason.

Second, when asked shortly before the end of the course, a majority of the PSP course participants claimed they would use PSP techniques for "larger" tasks, but not for small ones. If one is willing to believe this statement, it may be that we would have found larger PSP benefits if your experiment had used a larger task.

Third, and maybe most importantly, Watts Humphrey suggests that a working environment which actively encourages PSP usage is a key ingredient for PSP success. Our subjects were working alone and hence had no such environment.

Summing up, we conjecture that PSP training *alone* provides only a fraction of the expected benefits. Later encouragement towards actual PSP use appears to be necessary before large improvements are realized. Nevertheless, we believe that, in principle, the PSP is worthwhile and that our experiment provides support for this opinion.

Given the low degree of use of the actual PSP techniques despite a 15-week course, it appears a worthwhile research topic whether and how the PSP benefits can be obtained with a *much* smaller training program than the standard PSP course. We are pursuing such research and have already obtained first results [10]. Further, the experiment shows that we need to understand better how to make people *use* methods: what technical, social, and organizational means improve the level of *actual* use of a method as opposed to just the level of training or infrastructure provided?

Acknowledgements

We thank Oliver Gramberg for his contributions to our PSP course and Michael Philippsen for multiple thorough reviews of this paper and very helpful criticism. We thank the anonymous reviewers for their detailed and helpful comments. Most of all we thank more than fifty patient individuals who guinea-pigged the experimental setup or participated in the experiment itself.

Appendix: Raw result data

Data of the 24 successful participants of the PSP-trained group:

time	estim	A	reli	reliS	LOC	lang
3.8	5.0	2	100.0	98.4	258	Java
4.0	3.0	2	76.5	92.1	325	C
6.9	8.3	1	100.0	100.0	274	C++
7.3	5.0	1	100.0	100.0	174	Java
7.3	6.0	2	99.2	100.0	303	C
7.8	5.0	1	100.0	100.0	427	Java
8.2	3.5	2	0.0	89.5	199	C
9.7	6.0	1	100.0	100.0	433	Java
9.8	10.0	2	100.0	10.2	211	Java
10.1	18.0	1	100.0	98.4	226	C++
10.9	10.0	1	100.0	98.4	365	C
11.2	6.3	1	100.0	100.0	505	C++
11.9	5.2	2	100.0	100.0	287	C++
13.3	10.0	3	0.0	0.0	245	Sather-K
13.8	4.3	2	100.0	10.2	437	Java
15.2	4.0	5	100.0	98.4	274	C++
16.1	6.7	2	98.9	96.8	391	C
17.8	10.0	3	100.0	10.2	729	Java
19.1	12.6	2	100.0	98.4	334	C++
19.6	4.0	4	98.5	100.0	583	C++
19.7	12.0	3	100.0	100.0	467	Java
20.5	7.2	2	100.0	98.4	386	Modula-2
25.3	11.2	1	98.1	100.0	605	C++
27.6	11.4	4	98.9	96.3	323	Java

`time` is the actual work time in hours, `estim` the estimated work time. `A` is the number of acceptance tests needed. `reli` is the reliability on the standard data set, `reliS` is the reliability on the "surprising" data set. `LOC` is the length of the delivered program (excluding only empty lines), `lang` is the programming language used.

Data of the 16 successful participants of the non-PSP-trained group:

time	estim	A	reli	reliS	LOC	lang
3.0	3.0	1	99.2	98.4	168	C++
3.5	2.0	1	100.0	98.4	152	C++
3.8	8.0	1	100.0	10.2	249	Java
4.8	6.0	1	100.0	10.2	199	Java
5.0	4.0	1	100.0	98.4	267	Java
6.2	3.5	2	100.0	10.1	204	Java
7.1	5.0	1	99.2	98.4	396	Java
7.4	8.0	1	100.0	10.2	165	Java
7.5	10.0	2	0.0	1.1	309	C++
8.7	5.0	2	98.9	90.9	185	Java
13.0	5.0	3	100.0	10.2	422	Java
15.1	6.0	3	100.0	10.2	143	Java
18.1	5.0	3	100.0	100.0	355	Java
39.7	16.0	6	100.0	97.9	640	Java
48.9	6.7	2	100.0	10.2	667	Java
63.2	6.7	3	99.6	10.2	360	Java

Author Biographies

Lutz Prechelt worked as senior researcher at the School of Informatics, University of Karlsruhe, where he also received his diploma (1990) and his Ph.D. (1995) in Informatics. His research interests include software engineering (in particular using an empirical research approach), compiler construction for parallel machines, measurement and benchmarking issues, and research methodology. Since April 2000 he works for abaXX Technology, Stuttgart. Prechelt is a member of IEEE CS, ACM, and GI.

Barbara Unger graduated with a diploma degree in 1995 and is currently working as a PhD candidate in the Institute for Program Structures and Data Organization at the University of Karlsruhe. Her main research interests are in empirical software engineering with a focus on design patterns. She is a member of the IEEE and the IEEE Computer Society.

References

[1] Larry B. Christensen. *Experimental Methodology.* Allyn and Bacon, Needham Heights, MA, 6th edition, 1994.

[2] Bradley Efron and Robert Tibshirani. *An introduction to the Bootstrap.* Monographs on statistics and applied probability 57. Chapman and Hall, New York, London, 1993.

[3] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya. Introducing the Personal Software Process: Three industry case studies. *IEEE Computer*, 30(5):24–31, May 1997.

[4] W. Hayes and J.W. Over. The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[5] Watts S. Humphrey. *Managing the Software Process*. SEI series in Software Engineering. Addison Wesley, 1989.

[6] Watts S. Humphrey. *A Discipline for Software Engineering*. SEI series in Software Engineering. Addison Wesley, Reading, MA, 1995.

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

[8] Philip M. Johnson and Anne M. Disney. The personal software process: A cautionary case study. *IEEE Software*, 15(6):., November 1998.

[9] H.M. Parsons. What happened at Hawthorne? *Science*, 183(8):922–932, March 1974.

[10] Lutz Prechelt and Georg Grütter. Accelerating learning from experience: Avoiding defects faster. *IEEE Software*, 2000. Submitted April 1999.

[11] Lutz Prechelt and Barbara Unger. A controlled experiment on the effects of PSP training: Detailed description and evaluation. Technical Report 1/1999, Fakultät für Informatik, Universität Karlsruhe, Germany, March 1999. ftp.ira.uka.de.

[12] Lutz Prechelt and Barbara Unger. How does individual variability influence schedule risk?: A small simulation with experiment data. Technical Report 1999-12, Fakultät für Informatik, Universität Karlsruhe, Germany, September 1999. ftp.ira.uka.de.