Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 747–763
The Journal of Systems and Software
www.elsevier.com/locate/jss

Teaching disciplined software development

Dieter Rombach [a], Jürgen Münch [a], Alexis Ocampo [a,*], Watts S. Humphrey [b], Dan Burton [b]

[a] Fraunhofer Institute for Experimental Software Engineering, Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
[b] Software Engineering Institute, Carnegie Mellon University, 15213-3890 Pittsburg, USA

Received 1 April 2007; received in revised form 8 June 2007; accepted 9 June 2007
Available online 16 June 2007

Abstract

Discipline is an essential prerequisite for the development of large and complex software-intensive systems. However, discipline is also important on the level of individual development activities. A major challenge for teaching disciplined software development is to enable students to experience the benefits of discipline and to overcome the gap between real professional scenarios and scenarios used in software engineering university courses. Students often do not have the chance to internalize what disciplined software development means at both the individual and collaborative level. Therefore, students often feel overwhelmed by the complexity of disciplined development and later on tend to avoid applying the underlying principles. The Personal Software Process (PSP) and the Team Software Process (TSP) are tools designed to help software engineers control, manage, and improve the way they work at both the individual and collaborative level. Both tools have been considered effective means for introducing discipline into the conscience of professional developers. In this paper, we address the meaning of disciplined software development, its benefits, and the challenges of teaching it. We present a quantitative study that demonstrates the benefits of disciplined software development on the individual level and provides further experience and recommendations with PSP and TSP as teaching tools.

Keywords: Software development; Productivity; Defect density; Size estimation; Effort estimation; Yield; Personal software process; Team software process; Experimental software engineering; Software engineering education

1. Introduction

In this paper, we use a definition of discipline that relates to skill building. The "focus of discipline is on improving performance … it concerns the fidelity with which a defined process is actually followed" (Humphrey, 2006). Discipline is particularly important in software development because many software products are used in critical applications, and because undisciplined software development work has a large potential for causing economic or even physical harm. Over the last 20 years, a growing family of technical and management practices have been developed that, if properly used, will consistently deliver quality products on committed schedules. However, when team members do not properly follow these practices, their projects are typically late, over cost, and produce poor-quality products.

The Personal Software Process (PSP) course guides faculty in teaching disciplined development practices to software engineering and computer science students (Humphrey, 2005). While the PSP has not yet been widely adopted by academic programs, there is increasing industrial use and the results show that, when engineers are disciplined in their personal practices, their performance improves. This paper summarizes a study of the data gathered while training 3090 engineers. Most of the students were experienced engineers working for industrial software development organizations, and the instructors were either from the Software Engineering Institute (SEI) at Carnegie Mellon University or were trained by the SEI.

* Corresponding author. Tel.: +49 631 6800 2167; fax: +49 631 6800 1399.
E-mail addresses: Dieter.Rombach@iese.fraunhofer.de (D. Rombach), Juergen.Muench@iese.fraunhofer.de (J. Münch), Alexis.Ocampo@iese.fraunhofer.de (A. Ocampo), watts@sei.cmu.edu (W.S. Humphrey), dburton@sei.cmu.edu (D. Burton).

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.06.004

1.1. Why is discipline important for software organizations?

The performance of a development organization is determined by the performance of its engineering teams. Further, the performance of an engineering team is determined by the performance of the team members. Finally, the performance of the engineers is, at least in part, determined by the practices these engineers follow in doing their work.

While communication skills, native ability, intelligence, and experience have an unquestioned effect on engineering performance, this study shows that the predictability, quality, and productivity of a software developer's work can be measurably improved through training in disciplined personal practices. Furthermore, studies have shown that this improved performance at the personal level results in comparable improvements in team and project performance (Davis et al., 2003; McAndrews, 2000). These benefits are typically manifested by shorter development cycle times, fewer test defects, and reduced development and maintenance costs.

1.2. Why is discipline important for students?

Development work is becoming more challenging every year, and to succeed at this work, aspiring engineers must focus on building their personal capabilities. "Excellence starts with the individual. Achieving excellence is a constant struggle, principally because the world is changing. What was once considered excellent no longer is. This means that we must continually focus on improving our personal capabilities" (Humphrey, 2000). The critical need, then, is to understand how performance is evaluated and know what would constitute excellence.

While the performance of students is largely determined by their ability to get good grades, the technical proficiency of practicing engineers is not as significant in performance evaluations and promotions. Except for the occasional high- and low-performing exceptions, most graduate engineers are assumed to be technically competent. One of the major differentiators in industry is the engineer's ability to consistently and predictably produce quality results. When engineers follow the disciplines taught by the PSP, they can accurately plan their work, make responsible commitments, consistently meet their commitments, and produce high-quality results.

While these skills are important to engineering management, they are particularly important to practicing engineers. The reason is that when engineers consistently meet their commitments, their managers soon realize that they can manage themselves and still produce excellent results. Then, since managers are typically very busy people, they will largely trust these engineers to manage themselves, and they will continue to trust them for as long as the engineers continue meeting their commitments. Finally, as any experienced engineer will attest, the ideal engineering job is to be given an interesting and challenging assignment and to be trusted to manage the work yourself. That is the benefit of doing disciplined engineering work.

1.3. Why is discipline important for software education?

Today, a typical software education does not teach disciplined engineering practices. As a result, the most common experience in software development organizations is that their products are late, over cost, and of poor quality. This means that typical software professionals work long hours under severe schedule pressure and spend a large portion of their time fixing defective products. Few engineers like to have pizza at their desks for dinner, work on most weekends, and to stay late into the night fixing defects in test.

Most people prefer more balanced and satisfying careers. Software engineering, when done with proper discipline, can be rewarding. It involves teamwork, creating exciting and useful products, and having the satisfaction of seeing your own creations do what you intended them to do. It is potentially a great career. However, today, software engineering has a poor image and student enrolments are falling world wide, despite the demand for software professionals. To meet industry needs, and to have a growing and vibrant academic community in computer science and software engineering, the software development career must be made more attractive to potential students. This is another important reason to teach disciplined software development.

The rest of this article is structured as follows: Section 2 presents the main characteristics of disciplined software development; Section 3 presents details of the study based on data collected from 3090 engineers, who participated in PSP trainings. The study's main objective was to investigate the effects of disciplined software development on the engineer's ability to consistently and predictably produce quality results; Section 4 presents a collection of experiences and lessons learned from different institutions in the world who have used PSP and the TSP introductory course as part of their curriculum; Section 5 presents a set of recommendations for teaching disciplined software development based on the study's observations and the lessons learned from several institutions; Section 6 presents the conclusions of this article.

2. Disciplined software development

Developing software and software-intensive systems in a disciplined way requires a significant transition from craft-based to engineering-style development. While mature organizations widely aim at transitioning organizational structures and procedures towards engineering-style software development, it is also important to apply engineering principles on the level of individual developers. One of the reasons is that most methods applied in software development are significantly human-based. As a consequence, the

effectiveness of most methods depends to a large extent on the individual capabilities of the workforce. In order to focus skill building for disciplined development, it is necessary to understand the main elements of engineering-style software and system development.

2.1. Elements of engineering-style software development

The term engineering is often associated with the following: Planning is based on experience from the past (such as evaluated models), processes are predictable, project execution is goal-oriented and adheres to defined processes, projects are traceable and controllable, and learning and improvement cycles are established. These principles are widely accepted and established in traditional disciplines such as production engineering or mechanical engineering. In applying these principles to software development, one needs to consider the specifics of software (e.g., software is rather developed than produced, the effects of techniques depend on the development environment, software development involves many creative activities, data is less frequent and mostly of a non-parametric nature).

There are several approaches to applying engineering principles to software development, including the problem-oriented Quality Improvement Paradigm (QIP) and the solution-oriented Capability Maturity Model Integration (CMMI). Essential elements of disciplined software development that are addressed at varying degrees in these approaches include:

- Prerequisites for engineering-style development with respect to processes: defined processes, prediction models (with respect to effort, schedule, quality), analytical and constructive quality assurance processes throughout the whole lifecycle, understanding of the context-dependent aspects of key methods and techniques.
- Prerequisites for engineering-style software development with respect to products: adequate documentation, traceable documentation, evolvable architecture.
- Prerequisites for engineering-style software development with respect to management: adequate workforce capabilities and staffing, sufficient continuing education and training, guaranteeing sustainability of core competencies.
- Prerequisites for engineering-style software development with respect to organizational improvement: traceable quality guidelines, comprehensive configuration management, learning organization.

2.2. The role of empirical studies for disciplined software development

A major element of engineering-style software development is that organizations are able to understand the effects of key techniques and methods in their project and development environments. Empirical studies are an important means to determine such effects. Having evidence about the effects of methods in varying contexts significantly reduces the risk of process changes, especially the risk that results from introducing innovative technology into an organization. Performing empirical studies such as controlled experiments or case studies can be seen as process prototyping. Being able to evaluate techniques requires skills such as measurement and study design. Nowadays, these aspects are only addressed to a minor degree in current improvement models and skill schemas. It would be beneficial for the future to also address those aspects on the level of an organization and on the level of individual skills.

This article presents the application of one of the principles of engineering-style development: The article presents an evaluation of the effects of a combination of techniques that are applied in the context of PSP courses. The evaluation mainly consists of a replication that aims at increasing the significance of the evidence for the benefits of disciplined software development.

3. Benefits of disciplined software development for individuals

This section presents the definition and results of an empirical study oriented to confirm the assumption that disciplined software development at the individual level (represented by PSP) is of benefit for developers. The study was conducted by the Fraunhofer Institute for Experimental Software Engineering (IESE) jointly with the Software Engineering Institute (SEI).

3.1. Study description

PSP is a self-improvement process designed to help software developers to control, manage, and improve the way they work (Humphrey, 1995). It can help students plan better, track their performance precisely, and measure the quality of their products. A previous analysis performed by Hayes et al. (1997) on data obtained from the training of 298 engineers who took the PSP course at SEI demonstrated the benefits of PSP on estimation and planning, on the quality of the software, and on the quality of the work process. However, that study could not demonstrate an improvement or benefits regarding the developers' productivity. A replication of this study was performed by Wesslén (2000). The results of that replication confirmed the results of the Over and Hayes study, although it was performed with data from university students.

The motivation for the study presented in this article is based on Hayes and Over's study, and on previous experiences from process improvement programs where the use of a strategic plan for improving process development capabilities, based on the PSP approach, was considered to be of significant value for software engineers and for the software organization to which they belong (Ocampo et al., 1999; Ocampo, 2001). The purpose of this study is to examine the benefits of disciplined process management

as represented by PSP by extracting from the data of the PSP training of 3090 engineers figures about the impact of the improvement program of PSP on the developers' size estimation accuracy, effort estimation accuracy, defect estimation accuracy, and productivity, as well as on the products' defect density, and on the yield.[1]

This section provides a description of the context, the definition of the study's goal, and the steps we followed to systematically perform our study.

3.1.1. Study context

The data to be analyzed was collected in the PSP summary reports during the training of 3090 engineers. The engineers are professional developers, and the trainings were done at the Software Engineering Institute as well as at external locations. During the course, the students were given 10 exercises, which were mainly programs for statistical calculations. PSP has a maturity framework much like CMM (Paulk et al., 1993), which shows its progression in improvement phases, also called levels. Students complete their exercises while following the process attained at each PSP level. The PSP levels introduce the following set of practices incrementally:

PSP0: Description of the current software process, basic collection of time and defect data.

PSP0.1: Definition of a coding standard, basic technique to measure size, basic technique to collect process improvement proposals.

PSP1: Techniques to estimate size and effort, documentation of test results.

PSP2: Techniques to review code and design.

PSP2.1: Introduction of design templates.

PSP3: Introduction of the concept of cyclic development.

3.1.2. Study goal

The Goal/Question/Metric Paradigm (Basili and Weiss, 1984) was adapted in order to structure the study's goal (Briand et al., 1997). The GQM supports empirical studies on the specification of measurement goals. The GQM goal template used for describing the goal of this study looks as follows:

"Analyze the ⟨object⟩
for the purpose of ⟨purpose⟩
with respect to ⟨quality focus⟩
from the viewpoint of ⟨perspective⟩
in the context of ⟨context⟩"

The object defines the entity to be analyzed, i.e., PSP levels 0, 1, 2 and 3. The purpose describes why the object is analyzed, e.g., for characterization, evaluation, improvement, control, prediction, comparison. In this concrete study, the object was analyzed for the purpose of evaluation. The quality focus defines the object's attribute(s) to be analyzed, which in this case were size estimation, effort estimation, defect density, productivity, defect estimation, and yield. The perspective defines who the expected user of the outputs is, e.g., researcher, developer, manager, and/or customer. Finally, the context defines the setting where the analysis takes place. In this study, the setting corresponds to the training of 3090 engineers on PSP.

As a result, the goal of this study was stated as follows:

> Analyze the *data collected at the PSP levels (0, 1, 2, 3)*
> for the purpose of *evaluating performance differences of engineers*
> with respect to *size estimation accuracy/effort estimation accuracy/defect estimation accuracy/ yield/ defect density/productivity from the viewpoint of a researcher in the context of the PSP training of 3090 engineers.*

3.1.3. Hypotheses definition

The hypotheses investigated in this study are related to the expected benefits of PSP, i.e., more accurate estimation of size, effort, and defects, higher quality of the products represented by fewer defects, a higher yield, and an increase in productivity. We considered the hypotheses defined by Hayes et al. (1997) to be appropriate for our study's goal. Additionally, two hypotheses were added in order to investigate defect estimation and yield. The following is the detailed definition of each hypothesis.

H1: As engineers progress through the PSP training, their size estimates gradually grow closer to the actual size of the program at the end of the training. More specifically, with the introduction of a formal estimation technique for size in PSP level 1, there is notable improvement in the accuracy of the engineers' size estimates.

H2: As engineers progress through the PSP training, their effort estimates gradually grow closer to the actual effort expended for the entire life cycle. More specifically, with the introduction of a statistical technique (linear regression) in PSP level 1, there is notable improvement in the accuracy of the engineers' effort estimates.

H3: As engineers progress through the PSP training, the number of defects injected and therefore removed per thousand lines of code (KLOC) decreases. With the introduction of design and code reviews in PSP level 2, the defect densities of programs entering the compile and test phases decrease significantly.

H4: Productivity, expressed and defined in PSP by the number of LOC per hour spent, increases with a higher PSP level.

[1] Yield is the percentage of defects injected before the compile phase that are removed before the first compile (Hayes et al., 1997).

H5: As engineers progress through the PSP training, their yield increases significantly. More specifically, the introduction of design review and code review in PSP level 2 has a significant impact on the value of the engineers' yield.

H6: As engineers progress through the last four exercises of the PSP training, their defect estimates gradually grow closer to the actual number of defects removed for the entire life cycle.

3.1.4. Goal quantification

The GQM also supported our study on the derivation of measures as part of the study design. The metrics were extracted from the PSP summary report that engineers had to fill out for each exercise. Exercises 1A, 2A, and 3A corresponded to level 0. Exercises 4A, 5A, and 6A corresponded to level 1. Exercises 7A, 8A, and 9A corresponded to level 2. Exercise 10A corresponded to level 3. The GQM study's goal was quantified as presented in Table 1.

Please note that the same estimation formulas used for (Hayes et al., 1997) have been used in this study. Hayes et al. (1997) justifies the use of the estimation formula as follows: "This formula differs from the one used in the PSP training class. For the purpose of this study the Actual is subtracted from the Estimate so that underestimates result in a negative value and overestimates result in a positive value. In the training class the equation used is (Actual – Estimate)/Estimate." The formulas corresponding to defect density, productivity and yield are taken from the training (Humphrey, 1995).

3.1.5. Selected test

The test selected for this study was the repeated measures ANOVA (Girden, 1992). The purpose of ANOVA is to test for significant differences between means in different groups or variables (measurements), usually arranged by an experimenter in order to evaluate the effects of different treatments or experimental conditions, or combinations of treatments or conditions. This test helped us to observe the differences between the PSP levels with respect to size estimation accuracy, effort estimation accuracy, defect density, yield, defect estimation accuracy, and productivity.

3.1.6. Data acquisition

The most important characteristics under which the information was collected are summarized in Table 2.

The data for this study was collected by the Software Engineering Institute from PSP classes taught from 1994 through mid-2005. All but a few of the very early classes taught in 1994 and 1995 were taught by SEI-trained PSP instructors. Class size averaged 10.5 students, with most of the classes having between 6 and 15 students. A few classes were as large as 20 and a few as small as one student. Most of the classes, 283 of 293, were taught in industry to practicing software developers. Less than 4% of the data is from students in a university setting.

When collecting this volume of data from individuals as they work and producing the derived PSP measures on the various PSP forms, the question is how accurate the data and the derived measures are. Johnson and Disney did a study of PSP data collection in 1999 from 10 students who wrote 89 programs (Johnson and Disney, 1999). They found that when the data was collected on paper forms and the students manually manipulated the data and calculated the derived measures, many errors were committed. They found over 1500 errors of either: omission where a data value was not collected, calculation where a derived measure was incorrectly calculated, or transcription where a clerical error was made when transferring data from one form to another.

Two aspects of data collection used in this study address the problems cited by Johnson and Disney. The first one is

Table 2
Context vector

⟨Attribute, value⟩	
Number of developers	3090
Software products	PSP exercises 1A–10A
Experience of developers	Professional developers
Requirements guidelines	PSP script for planning
Design guidelines	PSP script for design
Implementation guidelines	PSP script for development
Communication platform	PSP scripts, templates, and forms
Review guidelines	PSP script for development
Validation guidelines	PSP script for development
Life cycle model	Waterfall, iterative
Organizational context	PSP training courses

Table 1
Goal quantification

Variable	Name	Value
Independent	PSP level	PSP 0, PSP1, PSP2, PSP3
	PSP exercise[a]	7A, 8A, 9A, 10A
Dependent	Size estimation accuracy (H1)	(Estimated LOC − Actual LOC)/Estimated LOC
	Effort estimation accuracy (H2)	(Estimated minutes − Actual minutes)/Estimated minutes
	Defect density (H3)	(1000 × Total defects removed/Actual new and changed LOC)
	Productivity (H4)	(LOC/hour)
	Yield (H5)	100 × (Defects removed before the compile phase/Defects injected before the compile phase)
	Defect estimation accuracy (H6)	(Estimated defect density − Actual defect density)/Estimated defect density

[a] This variable applies only for defect estimation accuracy.

the use of a tool by the students as they record their data, and second one is the review of each student's PSP data by a trained PSP instructor. Since early 1997, students have been recording their raw data in a tool provided by the SEI. This tool calculates all the derived PSP measures from the raw data the students enter. This eliminates calculation errors and transcription errors caused by manually transferring data from one form to another form. Errors of omission and consistency are primarily addressed by the PSP instructor.

When a student completes a program assignment, an instructor reviews the student's PSP data to check whether it is complete and accurate and whether the student followed the process, and provides feedback to the student on the work. This review is guided by a checklist where checks are done to make sure that all the raw data was collected and that the data is self-consistent. When problems are found by the instructor, the assignment is returned to the student to be corrected. The goal in the class is to provide this feedback before the student does the next assignment. Because of the grading workload in larger classes, two instructors were often used in the classes with 12 or more students.

3.1.7. Threats to validity

The fact that the collection sheets were consistent and completely filled in did not necessarily mean that trainees were conformant to the PSP level. We considered the following independent variables as threats to validity because they were not controlled at the time data was collected:

Complexity of the Exercises: The A series of exercises is mainly a set of programs to perform statistical calculations. Some of them seem to be more complicated to accomplish than others. This fact can influence final results of the studies.

Heterogeneity of Professional Experience: The data used for the study corresponds to 3090 engineers who took part in the training at the Software Engineering Institute (SEI) or at external locations. The data provided no information that could help us devise how experienced the programmers were. It is possible that more experienced programmers had better performance, impacting the indicators defined in the hypotheses.

Heterogeneity of Programming Languages: PSP does not mandate engineers to use a certain programming language, but advises engineers to use the language they master the best. The results of our observed indicators could be affected by the fact that some programming languages are more suitable for statistical calculations than others (because they have more utilities or libraries). Fortunately, we had information about the programming languages used by the engineers. We used this information for clustering the developers into those who programmed using object-oriented languages, those who used structured languages, and those who used other languages. This helped us in performing separate ANOVA tests, comparing them with the initial ANOVA tests (with all data), and drawing conclusions about the influence of this threat on our observations.

Training Adaptation: In 2002, the Software Engineering Institute adapted the PSP training and instructors were allowed to let students perform exercise 10A by following PSP 2.1. Instructors at other sites may have continued using PSP3. However, the data provides no means for differentiating them. We decided to cluster the data into engineers who took the training before 2002 and engineers who did it during and after 2002. As in the previous threat, we could then perform for each cluster separate ANOVA tests for each hypothesis and compare the results to the initial ANOVA tests (with all data).

3.2. Data validation

Data validation ensures the correctness and completeness of the collected data. In the context of the PSP training, correctness, completeness, and consistency can be checked against the guidelines defined in Humphrey (2005) for filling out the scripts, forms, templates, and standards for the PSP levels of the training.

The guidelines define the formulas to correctly calculate some fields, and the fields that must be filled out in order to say that the data was completely collected. Equally, the dependencies between forms, templates, and standards define the consistency of the data. We have performed queries and operations on the values of the summary reports with the purpose of checking if trainees did follow the PSP guidelines and complete the demanded exercises. Furthermore, box plots were used to search for outliers originating from either inconsistent or corrupted data collected on the PSP forms. We only used data from those, whose data for all 10 exercises were correct, complete, and consistent. This was necessary in order to observe the performance of engineers throughout the entire PSP training. For example, if one engineer included inconsistent or incorrect data in exercise 6A, then most probably the resulting trends would not be credible. Table 3 shows the number of engineers who remained after having their data validated with respect to correctness, consistency, and completeness.

3.3. Study results and interpretations

3.3.1. Size estimation accuracy

Fig. 1 shows that the distribution of size estimation accuracy gradually narrows, from level to level, with an exception between levels 2 and 3.

Table 3
Number of engineers who provided correct, consistent, and complete data

Size estimation accuracy	Effort estimation accuracy	Defect density	Productivity	Yield	Defect estimation accuracy
2149	1854	1636	2196	1589	218

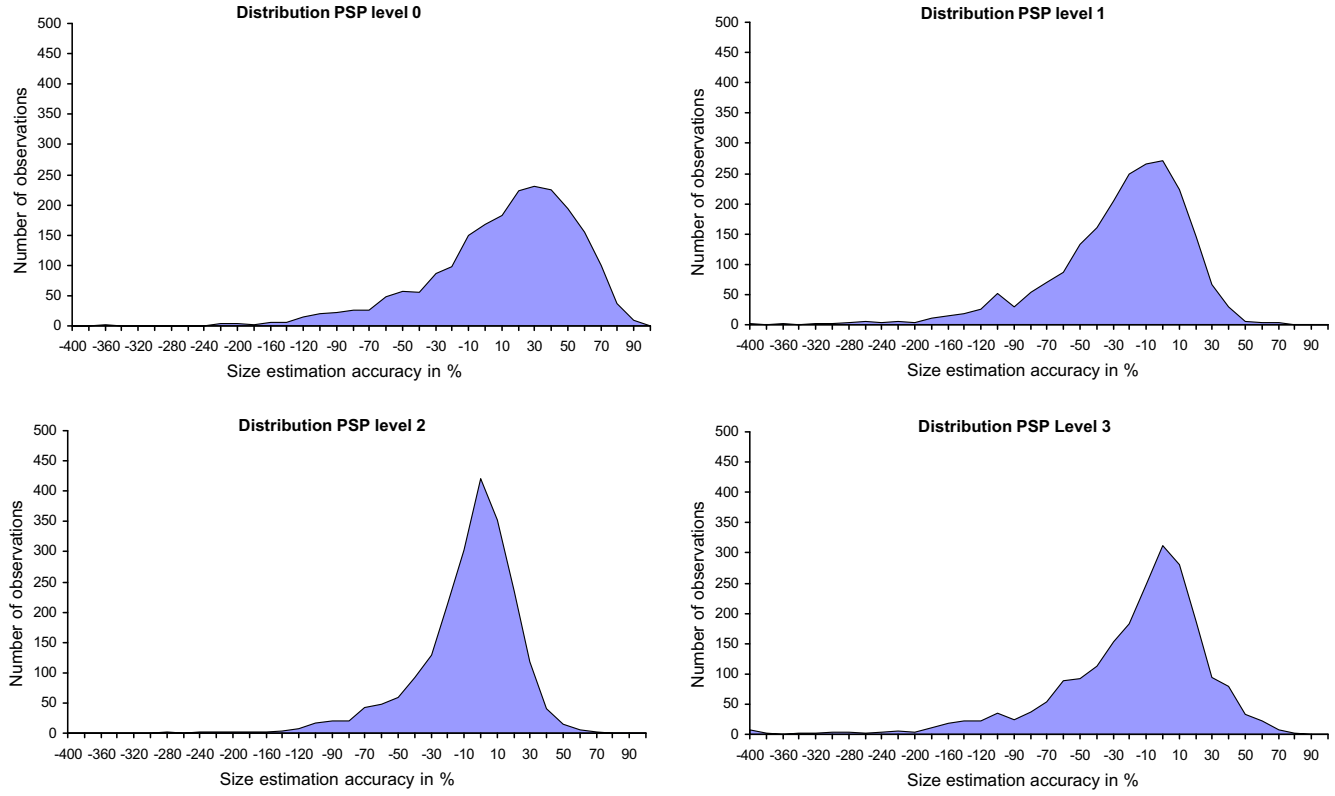

Fig. 1. Size estimation accuracy distribution.

The number of observations around 0 increases from level to level, and especially in the level 2, when engineers have more experience with the size estimation technique introduced in level 1.

Table 4 shows that the number of engineers per PSP level increases from level 0 to level 2, and then decreases from level 2 to level 3.

Fig. 2 shows the results of the repeated measures ANOVA for H1. The vertical bars denote the confidence interval of 95%, which means that the value of estimation accuracy is likely to be found between the upper and lower confidence limits. For each PSP level, we can observe the mean represented by the filled circle and the upper and lower confidence limits represented by the horizontal lines. The value 0% means that engineers estimated size perfectly. The expected trend should show a convergence towards 0%, which means that engineers incrementally improved their estimates. Fig. 2 shows a detriment of size estimation accuracy from level 0 to level 1, an improvement from level 1 to level 2, and another drop between levels 2 and 3, although not as strong as between levels 0 and 1. The repeated measures ANOVA revealed that the differences in size estimation accuracy across the four PSP levels are statistically significant (p-value < 0.01). Differences between adjacent levels are also statistically significant (p-value < 0.01).

We used the information on the programming languages used by developers, in order to look into this threat to validity more carefully. We did this by clustering the data containing all sorts of programming languages into object-oriented, structured, and other languages. Afterwards, we repeated the ANOVA test, and analyzed the

Table 4
Number of engineers whose size estimation accuracy (SEA) ended up ±20%

PSP level	0	1	2	3
SEA ±20%	730	910	1312	1029

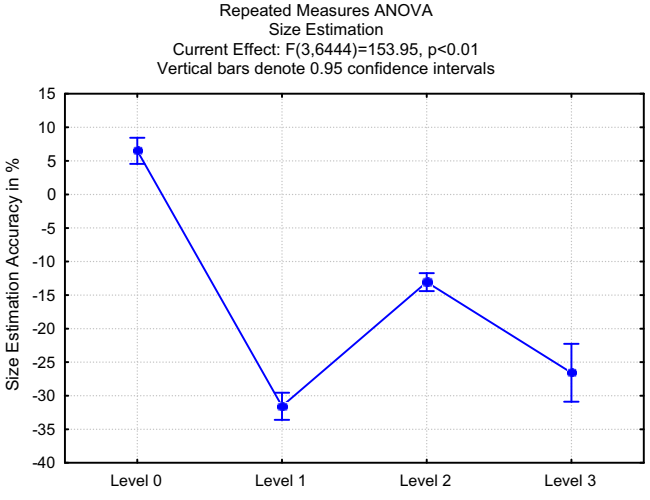

Fig. 2. ANOVA repeated measures for size estimation accuracy.

results. Fig. 3 shows the results for those developers who programmed their exercises using object-oriented programming languages. Similar trends as the one shown in Fig. 2 allow us to assume that this threat does not influence our interpretations.

We performed the same exercise for the threat to validity concerning the adaptation of the training. We did not find notable differences between the set of engineers trained before 2002 and the one trained after 2002. Therefore, we also assume that this threat does not affect our interpretations.

Interpretation: It may be useful to consider how the estimation process works for the interpretation. There are four levels or methods the engineers chose from, ranging from the most informal (method D; engineering judgment) to the most structured (method A; regression using object LOC estimates). When estimating is introduced in PSP1, engineers must start with one of the most informal methods (because they do not have enough historical data to use the most structured methods) and by the time they reach program 7 (the first PSP2 program), they have enough data that many can use the most structured method, method A. This is one reason why developers do not improve from PSP0 to PSP1 (i.e., because they start with the most informal method) and then show a considerable improvement in estimating from PSP1 to PSP2. We interpret the detriment from PSP2 to PSP3 in size estimation accuracy as an expected behavior, especially because developers need to incrementally build a credible set of historical data for their estimations. However, the difference is much smaller than between PSP0 and PSP1. In fact, when looking more carefully at each exercise, one can see how the curve tends to stabilize (see Fig. 4).

Based on the previous discussion and on the results of the repeated measures ANOVA across four levels, we see ourselves facing a dilemma concerning the acceptance or rejection of hypothesis H1. On the one hand, when looking at the ANOVA tests results, one may conclude that there is a detriment in general concerning size estimation. On the other hand, when looking at Fig. 1 and Table 4, one can see that the number of engineers whose estimation accuracy improved clearly increased from level 0 to level 4. Therefore, we neither accept nor reject the hypothesis and leave the issue to further research, meaning that data should be looked at in more detail and new tests should be performed.

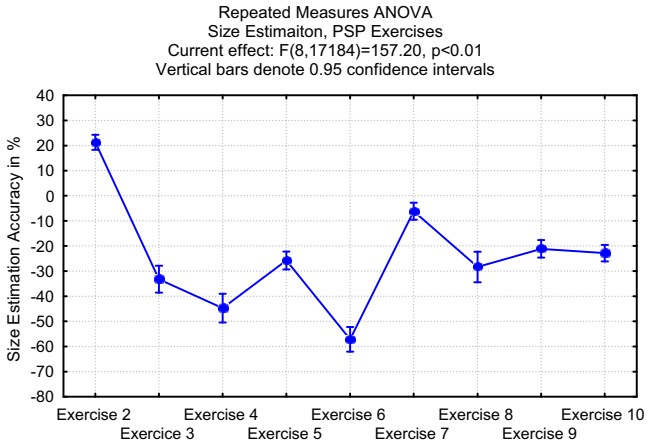

Fig. 4. ANOVA repeated measures for size estimation accuracy – PSP exercises.

3.3.2. Effort estimation accuracy

Just like in the previous size estimation results, the value 0%, means that engineers estimated effort perfectly. The expected trend should show a convergence towards 0%, meaning that engineers incrementally improved their estimates. Fig. 5 shows that estimation accuracy decreases between PSP Levels 0 and 1 and increases and gets closer to 0% on the next levels. A drastic increase can be observed from PSP1 to PSP2. The results of the repeated measures

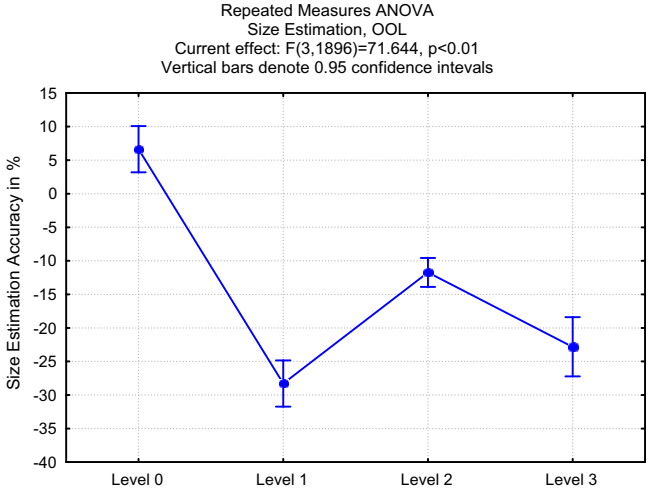

Fig. 3. ANOVA repeated measures size estimation accuracy OO programming languages.

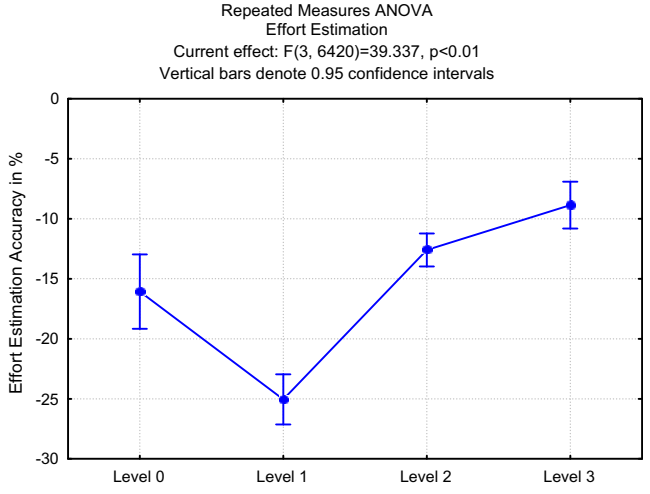

Fig. 5. ANOVA repeated measures for effort estimation accuracy.

Table 5
Number of engineers whose effort estimation accuracy (EEA) ended up being ±20%

PSP level	0	1	2	3
EEA ±20%	817	822	1107	967

ANOVA tests showed that the differences across (p-value < 0.01) and between adjacent levels (p-value(0–1) < 0.01; p-value(1–2) $= 0.01$, p-value(2–3) $= 0.01$) are statistically significant.

Table 5 shows the number of engineers with an effort estimation accuracy of ±20%. The number of engineers increased from level 0 to level 2, and decreased from level 2 to level 3, similar to what we observed for size estimation accuracy. We also observed that effort estimation accuracy increased more in level two because engineers had more experience with the technique introduced in level 1. Just like we did with the previous hypothesis, we analyzed the validity threats concerning the heterogeneity of programming languages and the training adaptation. We observed similar trends for each group of data: object-oriented, structured, and other programming languages, as well as for engineers trained before and after 2002.

Interpretation: The repeated measures ANOVA confirmed that the difference with respect to effort estimation between levels is significant. Especially after level 1, the trend shows a much clearer improvement in effort estimation accuracy than in the case of size estimation. In the PSP training, developers start to estimate effort in PSP1 using the estimated size as basis. Considering that effort estimation depends on size estimation, one can observe how smaller improvements in size estimation result in larger ones regarding effort estimation by looking at Figs. 3 and 5. Hypothesis H2 claiming that those engineers who took their PSP training improved with regard to effort estimation can be accepted based on the results of the statistical tests and reinforced by the previous observations.

3.3.3. Defect density

The repeated measures ANOVA was applied to test the significance of differences between PSP levels with respect to defect densities obtained in the compile and test phase as well as in overall development (see Figs. 6a, 6b, and 6c).

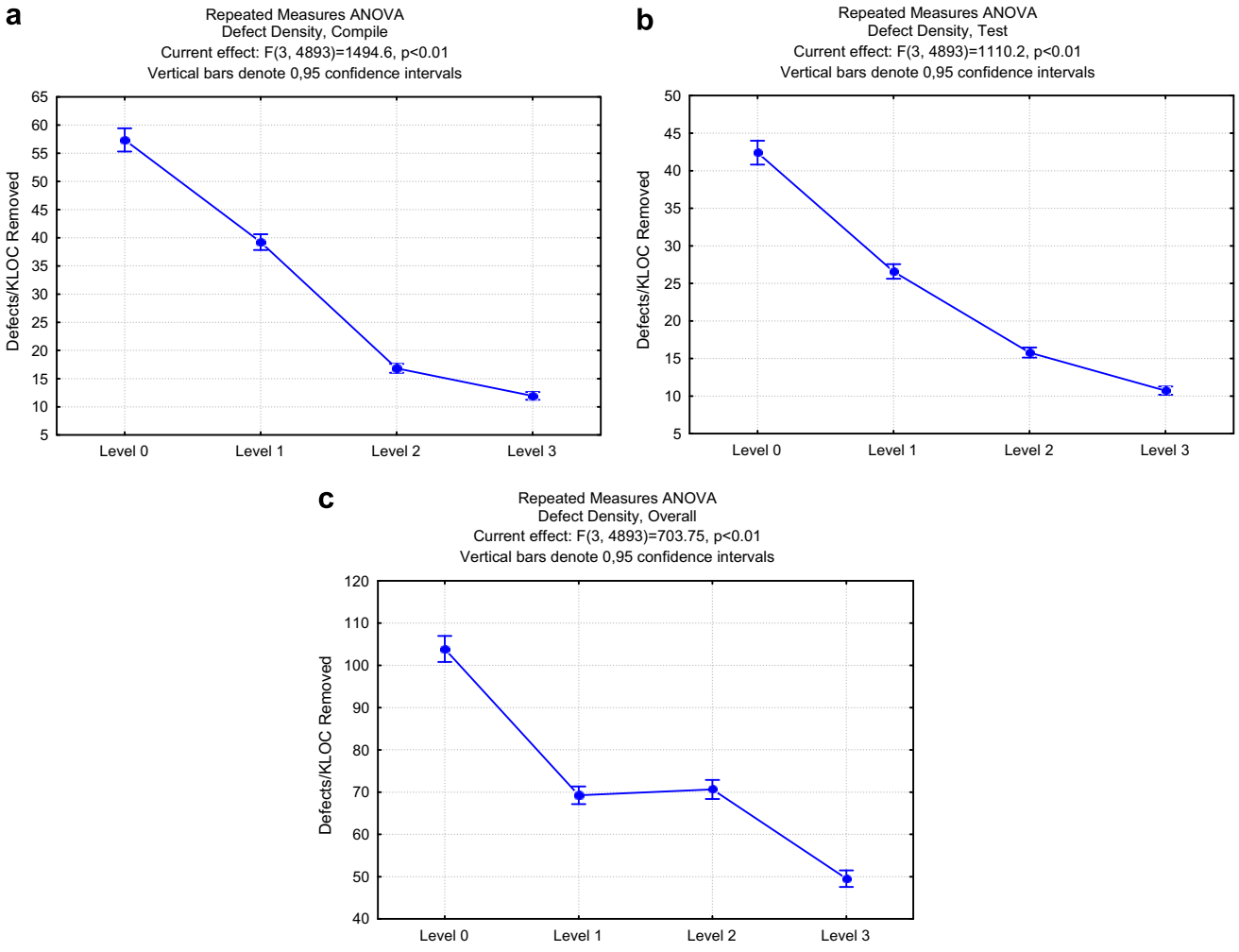

Fig. 6. ANOVA repeated measures for defect density (compile phase, test phase, and overall).

We observed similar and significant decreasing trends concerning defect density for the compile and test phases (p-values < 0.01). In the case of overall defect density, a general decreasing trend was observed with the exception of the trend between levels 1 and 2, where we found no significant difference (p-value $= 0.93797$). As in the previous studies, we evaluated threats to validity such as the heterogeneity of programming languages, and training adaptation. We observed similar trends, which allowed us to assume that these threats do not influence our interpretations.

Interpretation: In this case, there are two places where a defect density decrease can be seen, i.e., from level 0 to level 1, and from level 2 to level 3. We assume that the first one is due to the impact produced after engineers see their individual results for the first time, i.e., how many defects they have introduced. The second one confirms the hypothesis that after PSP level 2, where design and code reviews must be followed, developers introduce fewer defects. In general, a reduction of defect density can be observed in all cases. A reduction in total defect density translates directly into a reduction of the amount of rework for a software development organization. The results of the statistical tests and the previous observations support us in confirming hypothesis H3, which proposes a decrease of defect density in the compile and test phases, as well as for the entire life cycle.

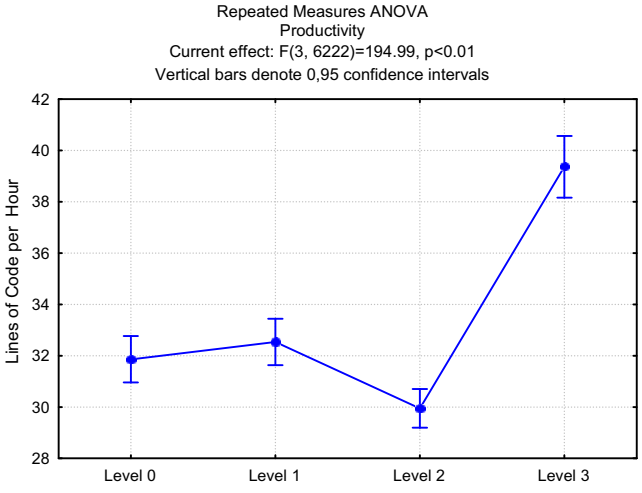

Fig. 7. ANOVA repeated measures for productivity.

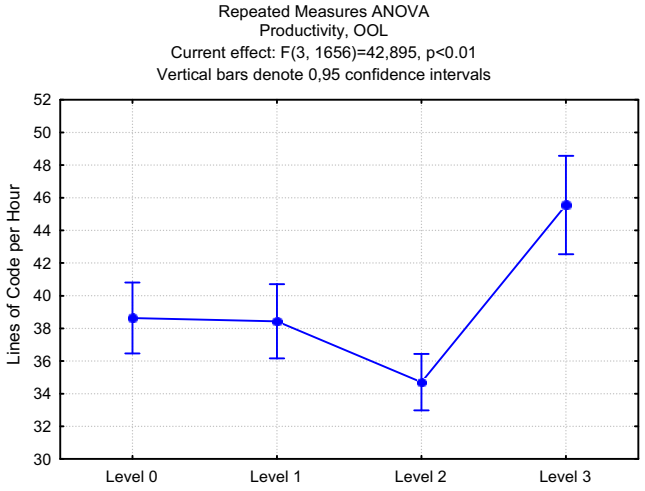

Fig. 8. ANOVA repeated measures for productivity object-oriented languages.

3.3.4. Productivity

Fig. 7 shows the results of the repeated measures ANOVA for H4.

The results of the repeated measures ANOVA tests showed a significant difference across PSP levels (p-value < 0.01). Between adjacent levels, differences are significant between 1 and 2 (p-value < 0.01), and 2 and 3 (p-value < 0.01), with the exception between 0 and 1 (p-value $= 0.062$). Examining the averages, we can see that there is an increase in productivity between levels 0 and 1, a reduction between levels 1 and 2, and a drastic increase between levels 2 and 3. The drastic increase from level 2 to level 3 called our attention, especially because the 10th

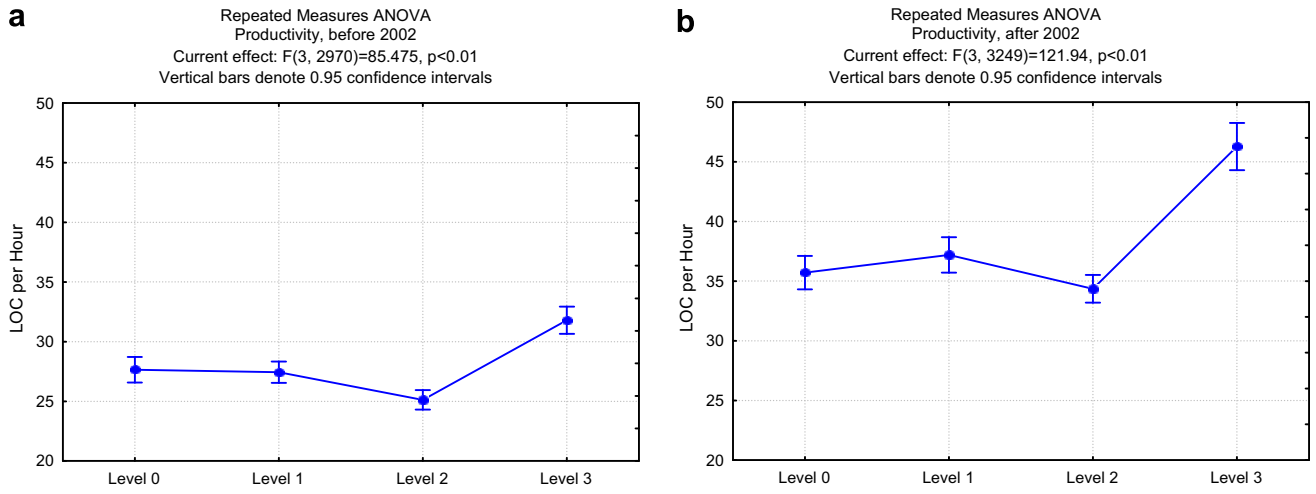

Fig. 9. ANOVA repeated measures for productivity before and after 2002.

exercise is one of the most demanding ones of the series, and because the PSP process demands quite a large set of information from the students. Concerning the programming languages' threat to validity, in all of the cases, i.e., object-oriented, structured, and other programming languages, we obtained similar results: the difference across levels is statistically significant (p-value < 0.01); the difference between adjacent levels is also statistically significant with the exception between levels 1 and 2 (p-value $= 0.786$); finally, a drastic increase from level 2 to level 3 was observed as shown in Fig. 8. Based on these results, we assume that the programming languages used by developers have no influence on our interpretation of the results.

We performed additional ANOVA tests in order to investigate the impact of the training modification on the interpretation. Figs. 9a and b show how in each case, i.e., before and after 2002, trends appear that are similar to the ones that correspond to the test performed for all data. The improvement in productivity after 2002 is higher than before 2002. We assume that this occurs because engineers got accustomed to working with PSP 2.1 and did not have to learn a new process, i.e., PSP3. In any case, we observed an important improvement.

Interpretation: The results of the ANOVA test showed that the difference in productivity across the different levels is statistically significant. Looking at the results concerning size estimation, effort estimation, and defect density, we could say that engineers benefit from PSP by improving these without losing on productivity. Furthermore, it is also known that exercise 10A is considered by PSP trainees to be the most demanding one of the series, which led us to assume that at this point engineers have mastered PSP so well that they can even increase their productivity. Based on the results of the ANOVA tests, and the previous analysis we accept the Hypothesis H4.

3.3.5. Yield

The repeated measures ANOVA revealed that the differences in Yield across the four PSP levels are statistically significant ($p < 0.01$). Differences are not significant between adjacent levels 0 and 1 (p-value $= 0.91$), and 2 and 3 (p-value $= 0.8$). A drastic increase in the yield can be observed between levels 1 and 2 (see Fig. 10). An increasing trend can be observed, especially accentuated between levels 1 and 2.

The shift to the right of the bars in the histogram confirms the ANOVA test (Fig. 11). The Yield increases, especially with the introduction of the design and code reviews in level 2. Threats to validity were treated in the same way as in the previous analyses. The statistical results showed similar trends, which leads us to assume that such threats do not significantly affect our interpretation.

Interpretation: We interpret that the increase in average yield from approximately 5% to nearly 55% from PSP level 1 to PSP level 2 is due to the introduction of formal code and design reviews in level 2. This is the major process change that occurs between these two levels. The result is

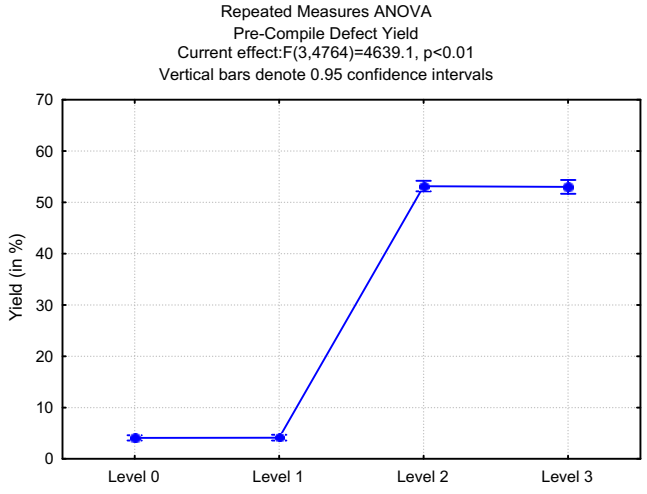

Fig. 10. ANOVA repeated measures for yield.

very similar to the one provided by Hayes et al. (1997), where the increase went from 10% to 55%. The non-significant difference between levels 0 and 1 can be explained in the same manner. Developers were not aware of the techniques or methods that they could use to remove defects injected before the first compile. An important observation is that the yield does not drop on the last level but holds, especially because the last exercises are quite complex. Finally, hypothesis H5 can be accepted because of the clear improvement that can be observed in the engineers.

3.3.6. Defect estimation accuracy

The results of the repeated measures ANOVA tests showed a significant difference across the PSP exercises (p-value < 0.01). Differences are significant between adjacent exercises 8A and 9A (p-value < 0.1), and between 9A and 10A (p-value < 0.01), with the exception between exercises 7A and 8A (p-value $= 0.1$).

The number of engineers whose estimation ended up being $\pm 20\%$ increased slightly from 7A to 9A and then slightly decreased.

Interpretation: The procedure introduced in PSP for estimating defects differs from the one for estimating size. Defects are estimated based on the historical data of defects per LOC. Developers underestimated defects after the new method was introduced, then got closer to 0, and finally overestimated. We deduce by looking at the trends of size and effort estimation that these fluctuations are normal while developers master the new method. Looking at the improvement in the defect density and yield results, we observe that developers have more control of the product's quality as they progress through the training. We assume that this can have a positive impact and result in more accurate defect estimations. Finally, we observe a similar case as the one observed for hypothesis H1. On the one hand Fig. 12 shows fluctuations without a clear improvement trend. On the other hand, we observe in Table 6 that the number of engineers that improve their

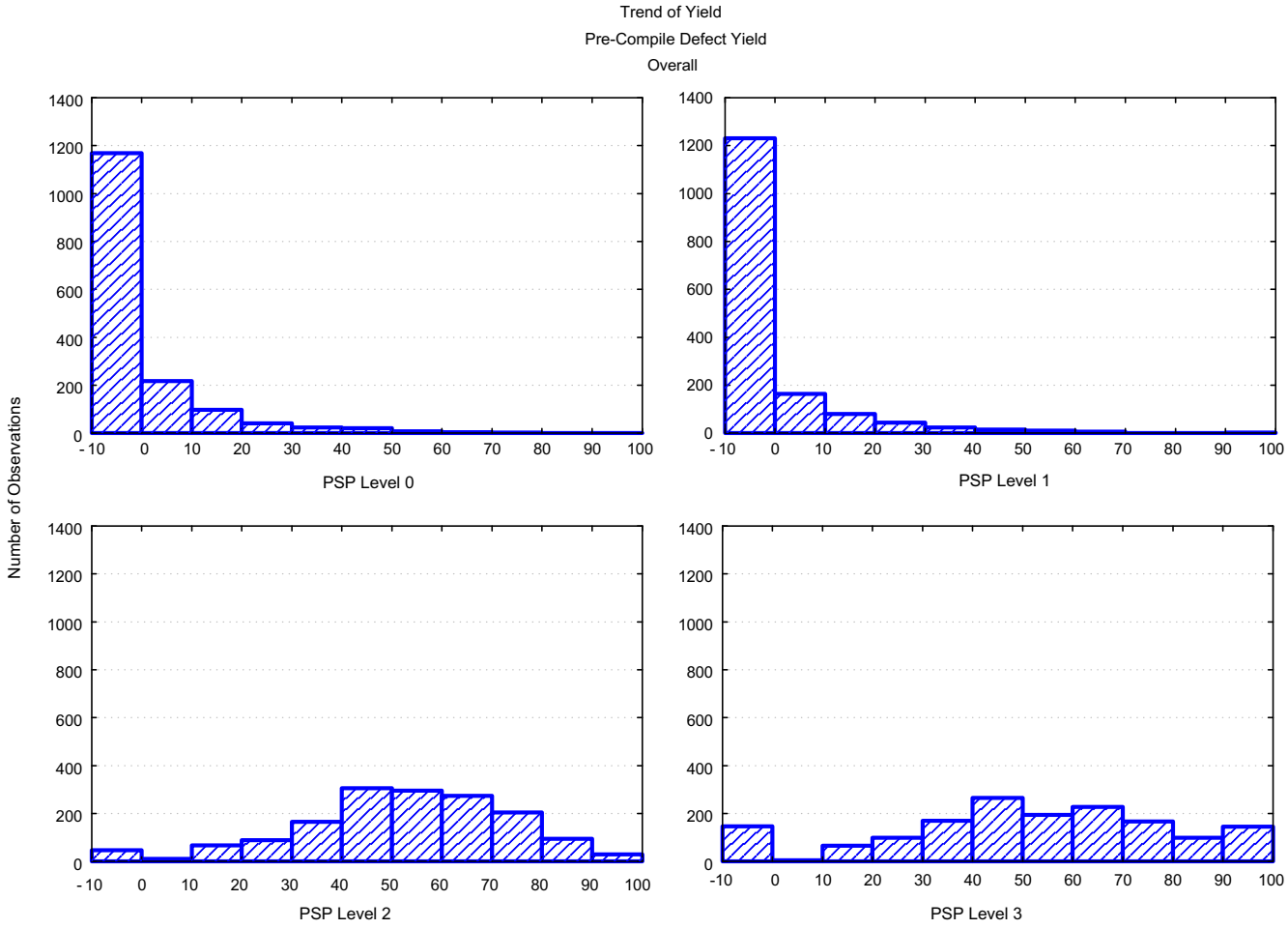

Fig. 11. Zoom in the overall group trend.

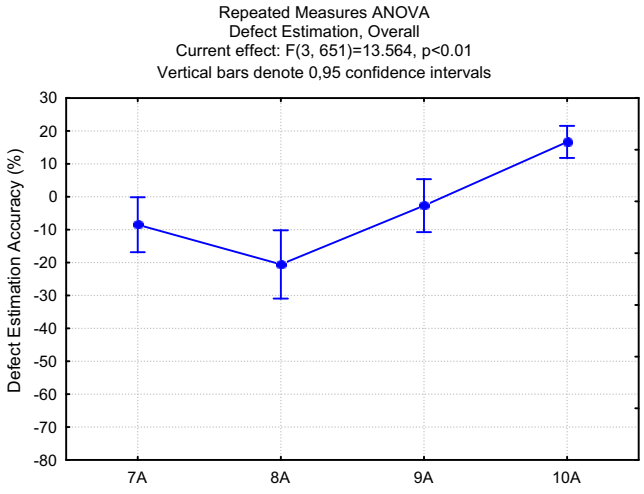

Fig. 12. ANOVA repeated measures for defect estimation accuracy.

Table 6
Number of engineers whose defect estimation accuracy (DEA) ended up being ±20%

PSP exercise	7A	8A	9A	10A
DEA ±20%	65	73	88	72

estimation increases. Therefore, we neither accept nor reject the hypothesis and leave the issue to further research.

4. Experiences with PSP and TSP as teaching tools

In the following section, we present some observed commonalities at various locations around the world that characterize the teaching of PSP and TSP introductory courses in universities, together with impressions from the teachers and students about their experience with PSP/TSP as techniques for building self-management skills. We have done this based on information described in Börstler et al. (2002), Carrington et al. (2001), Casallas et al. (2005), Hilburn and Humprey (2002), Johnson and Disney (1999) and Wesslén (2000), and additional interviews with instructors.

4.1. Observed commonalities in PSP applications

Students' level: Universities that teach PSP usually introduce it in computing science or information system programs either in early introductory programming courses as a complementary technique, or in 1–4th year software engineering courses. This introduction to PSP is usually

based on what Börstler et al. (2002) call *PSP-lite*, which is a book especially written for teaching first-year students (Humprey, 1997). A more PSP-oriented course called *PSP-full* (Börstler et al., 2002) based on Humphrey (1995) is preferred for senior students or post-graduate courses, since it is intended for professional software development. According to (Hilburn and Humprey, 2002), more than 30 institutions offer introductory and graduate PSP courses around the world.

Programming experience: Many institutions introduce PSP after students have learned how to program in previous courses (Börstler et al., 2002; Carrington et al., 2001). The idea behind this is not to have students concentrate on the peculiarities of the language more than actually needed and to have them concentrate more on PSP tasks. In cases where PSP is offered as an optional activity, students usually concentrated more on the programming tasks (Börstler et al., 2002).

Exercises: (Humphrey, 1995) proposes two series (i.e., A and B) of 10 exercises each for applying the concepts learned during the PSP training. However, ten exercises seem a bit too much for academic terms and for the "dynamics of typical student populations" (Hilburn and Humprey, 2002). Therefore, many institutions prefer to re-use their own exercises (Carrington et al., 2001). They are usually small to medium-size exercises. In some cases where PSP is taught in parallel, the exercises are designed to highlight programming topics. Other institutions demand between 7 and 9 of the PSP's exercises, but not all of them (Johnson and Disney, 1999; Wesslén, 2000).

Content adaptation: Those institutions that teach the PSP-*lite* version and integrate it into their curriculum usually prepare special lectures (2–5) for introducing the concepts, and tutorials for coaching students on how to use the forms, tools, and materials that PSP provides (Börstler et al., 2002). Such lectures are introduced early in the semester, so that students can apply their PSP knowledge during their programming exercises. Regarding the PSP sequence, important alterations are reported in Börstler et al. (2002) and Carrington et al. (2001), where the quality management techniques (e.g., defect collection, code/design reviews) are introduced as the first part of PSP instead of as the second part as advised by Humphrey (1995). The rationale behind this is that these techniques would provide more immediate and impressive results for students instead of size and effort estimation techniques. The institutions that follow the structure of PSP usually shortened the number of exercises and leave few lectures for postmortem analysis and students' reflection.

Tool support: Most institutions use automatic mechanisms for collecting the data and assuring their quality. Some institutions adapt the forms provided by PSP and develop their own Excel sheets or tools. Examples of such tools are: Leap, Pase, PSP Studio, and PSP Toolkit (Carrington et al., 2001). The other variant is to encourage students to collect the data manually, i.e., to copy and use the forms provided by the *PSP-lite* or *PSP-full* books.

Students' reported impressions: In general, the following remarks can be found in Börstler et al. (2002), Carrington et al. (2001), Johnson and Disney (1999), Wesslén (2000) and Hilburn and Humprey (2002) regarding the reactions of students who learned PSP. Students felt that they gained important knowledge in software engineering and that it was an important preparation for future jobs. This was in part because they felt more aware of their programming skills and shortcomings. Students recommended separating courses for learning a programming language from PSP lectures. They felt that learning both at the same time could be too much work. A typical reaction that confirmed this was shown by students in early stages of their education. They had to concentrate on both the programming language and PSP, and usually felt that PSP imposed an excessively strict process on them and that extra work would not pay off. Those students who used tool automation to collect PSP data showed more motivation to collect data, in contrast to those who did not have such tools.

Teachers' reported impressions: Teachers found PSP to be an effective mechanism for teaching software engineering practices to their students. They observed that immediate feedback is a powerful tool to motivate students. One mechanism to keep motivation high was to assume the role of coaches who monitor and control that students understand the principles well. Finally, most teachers agree that the potential benefits of PSP can only be obtained if careful adaptation to and integration into the actual curricula has been performed.

4.2. Observed commonalities in TSP applications

According to Hilburn and Humprey (2002), more than a dozen computing programs have used the introductory Team Software Process (TSPi) in software project courses. TSPi introduces techniques for successful teamwork and exposes students to more realistic software development tasks, which contributes to closing the gap between universities and industry. Please note that TSPi does not contain all aspects of the TSP that will need to be used for larger-scale industrial projects (Humphrey, 2000). The following is the result of identifying the most common features observed among academia while using TSPi.

Students' level and programming experience: Some institutions introduce TSPi during the second and third year of undergraduate courses on software engineering (Hilburn and Humprey, 2002; Casallas et al., 2005; Oktaba et al., 2003). At this point in time, students usually feel comfortable with a specific programming language and could be seen as accomplished programmers. Most institutions prefer object-oriented programming languages such as Java or C++, especially because these are part of their academic program (Hilburn and Humprey, 2002; Casallas et al., 2005; Oktaba et al., 2003).

Software project: Humphrey provides requirements for a software project to be completed in 15 weeks while applying TSPi (Humphrey, 2000). However, as in the case of

Casallas et al. (2005) and Oktaba et al. (2003), some academic institutions start with the original software projects and after a while replace them with their own projects.

Content adaptation: The TSPi course has been designed to be deployable in a normal university semester. It is suggested to last 15 weeks covered by two to three cycles in which a group of students participate in a software development project. The group of students should go through the following phases in each cycle: Launch, Strategy, Plan, Requirements, Design, Implementation, Test, and Postmortem. It is reported that institutions usually deploy TSPi in two cycles (Oktaba et al., 2003; Casallas et al., 2005; Hilburn and Humprey, 2002) because three cycles seems to be too tight and leave little time for discussing and analyzing the results. Some institutions avoid the use of the techniques proposed by TSPi for requirements and design and propose the use of others, such as techniques described in the Rational Unified Process (RUP) (Gornik, 2004).

Tool support: Aside from the forms and materials provided in the TSPi course, a tool for supporting the university courses is offered.[2] Institutions usually use this tool in the first trials and then customize it for subsequent courses (Oktaba et al., 2003). More sophisticated solutions consist of providing a complete virtual learning environment as a platform to support active collaborative learning as in the case of Casallas et al. (2005), where students develop their project guided by such a platform. Apart from providing the students with the required information about the TSPi process and theory, the platform provides an "all in one place" perception to the students by supporting team organization and communication activities with private spaces and planning and tracking tools.

Students' reported impressions: According to (Hilburn and Humprey, 2002), about 75% of the students who learned PSP were "positive about TSPi". Similar reactions can be seen from students in the cases of Casallas et al. (2005) and Oktaba et al. (2003). Students most liked the fact of having a very clear process and role descriptions that supported them in executing their tasks. Students also saw more benefits in the TSPi course after having attended PSP courses. They recognized the importance of TSPi by the end of the course with respect to planning and performance tracking, although they complained concerning the effort needed for collecting data.

Teachers' reported impression: In general, teachers underlined as the most important TSPi strength the capability that it gives to students to organize themselves in a very straightforward way. The central axis of TSPi is teamwork, which is achieved by the interdependence and interaction described in the TSPi scripts. Additionally, the TSPi course is a very good means for teaching a project course where software engineering must be applied. Teachers also considered it feasible to adapt TSPi to the unique characteristics of their institution.

[2] TSP Tool. Available at: http://www.sei.cmu.edu/tsp/tsp.html, November 2006.

5. Recommendations for using PSP and TSP as discipline drivers in education

Software engineering or project development courses should build up the self-management skills of students in a way that they can perform well in their future real jobs. However, barriers such as lack of motivation to follow a structured process, teamwork reluctance, and lack of commitment to quality impact the chances of success of such courses (Casallas et al., 2005). Lack of motivation to follow a process is attributed to the fact that accomplished programmers do not see the need for doing things differently, because they already think they know how to do them right. Teamwork reluctance is based on distrust in others and sometimes in oneself. Students tend to finish their work without having to depend on others. Lack of commitment to quality stems from the fact that usually, the final program delivered by the students to their teachers will not have any impact on other systems or affect anybody.

The results shown in this and previous studies regarding the impact of PSP on an engineer's performance and the experience of universities with PSP/TSPi give us confidence in assuring that both of them are suitable mechanisms to overcome such barriers. The following are our suggestions to take into account when using PSP/TSPi to build up the self-management skill of potential software engineers.

5.1. Customization of PSP and TSPi courses

Several authors underline the importance of customizing either the PSP training or the TSPi course to the context of the faculty. This can be accomplished by customizing the exercises, and adapting the contents and/or the tool support. However, please note that only those institutions that have tried PSP or TSPi several times and whose faculty was familiar with such technologies dared to do this customization. This means that customization has better chances of success if faculty members are confident about the goals and PSP/TSPi processes (Hilburn and Humprey, 2002; Carrington et al., 2001; Börstler et al., 2002). Institutions that plan to introduce either PSP or TSPi are advised to iteratively and incrementally customize the courses. This means that they can start with the original trainings and modify them incrementally as they acquire experience with them.

5.1.1. Exercises

It seems easier for institutions to use or adapt their own predefined project course exercise(s) instead of using the project defined by default for TSPi or PSP in Humphrey (2000) and Humphrey (1995), respectively. However, institutions must very carefully design such exercise(s), trying to cope with the goals of the "by default" exercises, i.e., "to give students team experience in developing intermediate-

sized software products'' in the case of TSPi or ''to provide students with experience and with data on their own disciplined software methods'' in the case of PSP. The exercises should be adapted to the students' current programming skills and should try to avoid the introduction of new technologies that could deviate the students' attention from the goals intended by PSP/TSPi.

5.1.2. Contents adaptation

One particular adaptation of the PSP process consists of first teaching students how to collect defects and understand the importance of quality management, and afterwards introducing the concepts related to size and effort estimation (Börstler et al., 2002; Carrington et al., 2001). Also, the use of the *PSP-lite* version seems more adequate than the *full* one for those institutions short on resources. In the case of TSPi, a common pattern was that the maximum number of cycles to be followed in the semester was two. The time needed by the students to reflect about what they did, how well they did, and what they learned is a relevant issue (Casallas et al., 2005; Hilburn and Humprey, 2002) that seems to be easier to accommodate in one semester if the number of cycles is two.

5.1.3. Tool Support

The availability of a comprehensive tool as in the case of Casallas et al. (2005) was cataloged as a very important motivation for following TSPi when accomplishing the assigned software project. In some cases, the existent PSP/TSPi tools were used in the first trials and then modified or replaced by a new tool that better fit the students' and the faculty's needs. Creating a comprehensive tool means connecting the PSP/TSPi material with the laboratory setting and the classroom as suggested by Hilburn and Humprey (2002) and implemented by Casallas et al. (2005). Such a tool should facilitate: data collection, data validation, communication and discussion among team partners, access to process material (forms, templates, scripts, tools), visualization of performance indicators (for the group and the individual), visualization of previous experiences (e.g., previous interviews with students in video format), defect management, and configuration management. Additionally, for non-English speaking institutions, translation of specific materials could facilitate the students' work. We recommend starting with a tool that provides basic functionalities, e.g., data collection, data validation, and communication among team partners, and that can be enhanced incrementally.

5.2. Integration of PSP as part of TSPi

The recommendation per-default is that PSP should be taught before TSPi (Humphrey, 1995). The reason is that students should really master techniques such as collecting data, estimating size, and effort, analyzing their results, and proposing self-improvement, before assuming a role in a software development group and working following the TSPi process. However, some institutions do not have the infrastructure and resources for following this advice. In those cases, other possibilities exist such as to either integrate them, or customize and teach them separately.

We consider it reasonable to integrate and teach PSP basic concepts inside TSPi, because several PSP tasks are demanded when following the TSPi process (for example, time and defect data collection, or size and effort estimation). These concepts can be introduced by demand in the TSPi-oriented course. In this way, students can learn important basic individual activities in the context of a project team reaching a common goal. TSPi effectively combines elements of collaborative learning, creating a more enjoyable environment for students and increasing their motivation (Casallas et al., 2005; Oktaba et al., 2003). In this way, the risks associated with the lack of motivation of students and faculty can be minimized. Teaching PSP or TSPi separately is advised for those institutions with enough resources for customizing the training in a goal-oriented and incremental manner. The institution might be aware of the risks and might manage them. For example, we observed that the experience of some institutions with PSP showed some frustration of the students, because of very intensive paper work in addition to quite complex programming exercises within a short period of time (Börstler et al., 2002).

6. Summary and outlook

This paper has been written with the purpose of underlining once more the importance for industry of disciplined software development at both the individual and organizational level and the responsibility that academia carries in making this possible. The main rationale behind our belief lies in the fact that self-organized developers are a source of personal and team satisfaction, and that this is only possible if potential engineers (i.e., students) are taught to discover, use, and manage their personal capabilities as part of a team with a common goal.

We argue that PSP and TSP are a good means to introduce discipline concepts to potential engineers, and that they provide a basic and useful training framework. We have shown the benefits at the individual level, based on a replication and extension from a previous study, where we measured the impact of PSP as a representative of disciplined software development with respect to the developer's size, effort, and defect estimation accuracy, to the quality of products, the yield, and the productivity.

We observed how, despite heterogeneous data sets (i.e., different years, different programming languages, and different programming experience), the results continue to show benefits for engineers.

We have also summarized and compared several university experiences, looking for patterns regarding teaching contents, students' reactions, and faculty reactions. The feeling that PSP and TSPi are good means for teaching software engineering at the personal and group level is

generally present among the institutions studied. We extracted a set of recommendations based on such experiences and the quantitative study that could be used as starting points for those interested. The following are topics that we consider important for future research work:

Size estimating accuracy should be more concerned with balanced estimates than percentage error. We have observed that developers always have some error in their personal estimations. However, if developers made balanced estimates, they would underestimate as often as they overestimated and as a consequence, when they work as part of a team, the total team's total estimates would be more accurate. We find this is the case with the TSPi. A further study could examine the degree to which PSP students make more balanced estimates.

Another topic consists of investigating the relationship between productivity and effort estimation accuracy. With the estimation method introduced in PSP 1, i.e., Proxy Object Based Estimation (PROBE), new effort estimates are based on past performance (i.e., actual hours per estimated object lines of code). Thus, when productivity improves, effort estimates should have a consistent bias toward underestimates. This bias should not show up with size estimates. By examining size and effort estimating error, productivity, and the degree of process change, one could better understand the consequences of process change and learning.

There is also the overhead required for planning and process postmortem. With larger programs, the relative time required to plan a project and do the postmortem analysis should be less. The question to address with a further study concerns whether planning time and postmortem time are dependent on project size or whether they are more or less constant and could be viewed as overhead. In the latter case, the planning process could be made more accurate by allocating a relatively fixed amount of time for either planning or postmortem or both, and a percentage of time per phase for the rest of the work.

We observed the need to answer additional important questions such as: How will defect estimation behave in further studies? How could we prepare a set of exercises that allows us to separate the complexity of exercises from the PSP levels? To what extent is a virtual environment the most appropriate tool for teaching discipline teamwork? What kind of feedback is received best as motivation by the students: defects? size estimation? effort estimation?

Finally, we believe that through the constant observation of students during software engineering courses and the design and execution of empirical studies, a better understanding of the most appropriate possibilities for teaching disciplined software development can be achieved.

Acknowledgements

We would like to thank Marcus Ciolkowski from the Fraunhofer Institute for Experimental Software Engineering (IESE) for his valuable support during the execution of the study. We would also like to thank Anita Carleton and Jim Over from the Software Engineering Institute (SEI) for their important ideas during the definition of the study. Additionally, we thank Professor Rubby Casallas from Los Andes University (Colombia), Professor Hannah Oktaba from the National Autonomous University of Mexico (Mexico), and Professor David Carrington from the University of Queensland (Australia) for providing us with valuable input and sharing their experiences with us. Finally, we would like to thank Sonnhild Namingha from Fraunhofer IESE for preparing the English editing of this paper.

References

Basili, V.R., Weiss, D., 1984. A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering 10 (3), 728–738.

Börstler, J., Carrington, D., Hislop, G.W., Lisack, S., Olson, K., Williams, L., 2002. Teaching PSP: challenges and lessons learned. IEEE Software 19 (5), 42–47.

Briand, L.C., Differding, C.M., Rombach, H.D., 1997. Practical guidelines for measurement-based process improvement. In Software Process: Improvement and Practice 2 (4), 253–280.

Carrington, D., McEninery, B., Johnston, D., 2001. PSP in the large class. Proceedings of the Conference on Software Engineering Education and Training (CSEE& T). IEEE Computer Society Press, pp. 81–88.

Casallas, R., Osorio, L.A., Lozano, A., 2005. The challenge of teaching a software engineering first course. In: Proceedings of the fifth international workshop on Active Learning in Engineering, Delft-Amsterdam, The Netherlands.

Davis, N., Mullaney, J., 2003. Team Software Process (TSP) in Practice. SEI Technical Report CMU/SEI-2003-TR-014.

Girden, E.R., 1992. ANOVA. Repeated Measures. Sage Publications, Thousand Oaks, Quantitative Applications in the Social Sciences 84.

Gornik, D., 2004. IBM Rational Unified Process: Best Practices for Software Development Teams, Rational Software White Paper TP026B, Rev 11/01.

Hayes, W., Over, J.W., 1997. The Personal Software Processs (PSP): An Empirical Study of the Impact of PSP on Individual Engineers. Software Engineering Institute Technical Report, CMU/SEI-97-TR-001.

Hilburn, T.B., Humprey, W.S., 2002. Teaching teamwork. IEEE Software 19 (5), 72–77.

Humphrey, W.S., 1995. A Discipline For Software Engineering, re-printed ed. Reading, Addison-Wesley.

Humprey, W.S., 1997. Introduction to the Personal Software Process. Reading, MA, Addison Wesley.

Humphrey, W.S., 2000. Introduction to the Team Software Process. Reading, MA, Addison Wesley.

Humphrey, W.S., 2005. PSP: A Self-Improvement Process for Software Engineers. Reading, MA, Addison Wesley.

Humphrey, W.S., 2006. TSP: Coaching Development Teams. Addison Wesley, Reading, MA.

Johnson, P., Disney, A., 1999. A critical analysis of PSP data quality: results from a case study. Empirical Software Engineering 4 (4), 317–349.

McAndrews, D.R., 2000. The Team Software Process (TSP): An Overview and Preliminary Results of Using Disciplined Practices. Carnegie Mellon University Technical Report CMU/SEI-2000-TR-015.

Ocampo, A., Casallas, R., Soto, M., 1999. An implementation of the PSP in an industrial context: a case study. Proceedings of the 2nd European Software Measurement Conference (FESMA). the Netherlands, Amsterdam.

Ocampo, A., 2001. Looking for the path to the PSP enactment. Proceedings of the 4th European Software Measurement Conference (FESMA). Heidelberg, Germany.

Oktaba, H., Ibaguengoitia, G., 2003. Quality in Software Proceses. In: Piatini, M., García, F. (Eds.), TSPi Example (in Spanish). Desarrollo y Mantenimiento de Software. RAMA.

Paulk, M., Curtis, B., Chrisis, M.B., 1993. Capability Maturity Model for Software Version 1.1, Software Engineering Institute Technical Report, CMU/SEI-93-TR.

Wesslén, A., 2000. A replicated empirical study of the impact of the methods in the PSP on individual engineers. Empirical Software Engineering. 5 (2), 93–123.

Dieter Rombach is a Full Professor in the Department of Computer Science at the University of Kaiserslautern, Germany. He holds a chair in software engineering, is executive and founding director of the Fraunhofer Institute for Experimental Software Engineering (IESE), and chairs the Fraunhofer ICT group, which aims at shortening the time needed for transferring research technologies into industrial practice. His research interests are in software methodologies, modeling and measurement of the software process and resulting products, software re-use, and distributed systems. Results are documented in more than 180 publications in international journals and conferences. In 2000, he was awarded the Rhineland-Palatinate State Service Medal. In January 2003, the Institute of Electrical and Electronical Engineers (IEEE) elected him to the grade of Fellow for contributions to experimental Software Engineering. He heads several research projects funded by the German government, the European Union, and industry, and leads a federally funded project (VSEK) aimed at building up a German repository of knowledge about innovative software engineering technologies.

Jürgen Münch is Division Manager for Quality Management at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany. Before that, he was an executive board member of the temporary research institute SFB 501 "Development of Large Systems with Generic Methods" funded by the German Research Foundation (DFG). He received his Ph.D. degree (Dr. rer. nat.) in Computer Science from the University of Kaiserslautern, Germany. His research interests in software engineering include: (1) modeling and measurement of software processes and resulting products, (2) software quality assurance and control, (3) technology evaluation through experimental means and simulation, (4) software and system product lines and (5) technology transfer methods. He has been teaching and training in both university and industry environments, and also has significant project management experience. He is a member of IEEE, the IEEE Computer Society, and the German Computer Society (GI), a member of the program committee of various software engineering conferences, and has published more than 60 international publications. He has been program co-chair of the Profes 2006 and Profes 2007 conferences and will serve as program co-chair for ESEM 2008.

Alexis Ocampo received his Masters degree in Systems Engineering and Computation from Los Andes University, Colombia, in 1999 and his title as Systems Engineer from the Industrial University of Santander, Colombia, in 1997. Since February 2002, he has been a research scientist at the Fraunhofer Institute for Experimental Software Engineering (IESE) in Kaiserslautern, Germany, in the Department of Processes and Measurement. Before that, he worked for 5 years as a research developer on new technologies and methodologies with the software company Heinsohn Associates, Bogotá Colombia. His master thesis entitled "Implementation of PSP in the Colombian Industry: A case study" was developed within this company. He also worked as an instructor at the University of Los Andes in the Department of Systems and Computation. His research interests in software engineering include: (1) modeling and measurement of software processes and resulting products, (2) software quality assurance and control and (3) technology transfer methods.

Watts Humphrey received his B.Sc. degree in Physics from the University of Chicago in 1949 and an MS in Physics from the Illinois Institute of Technology in 1950. In 1951, he obtained an MBA degree from the University of Chicago. He worked for Sylvania Electric Products, Inc. in Boston, MA from 1953 until 1959 on computer design. From 1959 until 1986, he was with the IBM Corporation in Armonk, New York. Among other jobs, he was Director or Programming, Director of Systems and Application Engineering, and Vice President of Technical Development. In 1986, he joined the Software Engineering Institute, where he was Director of the Process Program and is now an Institute Fellow. He has been a member of the Malcolm Baldrige National Quality Award Board of Examiners and is on the editorial board of numerous technical journals. He has published 11 books and numerous papers. In 1998, he was granted an honorary Ph.D. in Software Engineering by Embry Riddle Aeronautical University. In 2000, the Watts Humphrey Software Quality Institute in Chennai, India was named in his honor and, in a White House ceremony, the President of the United States awarded him the 2003 National Medal of Technology.

Dan Burton received his Bachelor of Science degree in Electrical Engineering from Carnegie Mellon University in 1967 and a Master of Science in Electrical Engineering from the US Air Force Institute of Technology in 1974. From 1968 through 1988, he was an officer in the US Air Force and served in numerous software engineering positions. In his last position in the Air Force, he was a member of the government team that established the Software Engineering Institute at Carnegie Mellon University and then headed up the on-site liaison office. From late 1988 through mid-1991, he worked at Tartan Labs, where he led the development of an Ada compiler for a digital signal processor. In mid-1991, he rejoined the Software Engineering Institute as a member of the technical staff. Since 1995, he has been working with Watts Humphrey on the Personal and Team Software Process.