Software Engineering Institute

TSP Symposium 2012 Proceedings

William Richard Nichols
Álvaro Tasistro, Universidad ORT Uruguay
Diego Vallespir, Universidad de la República
João Pascoal Faria, Faculty of Engineering, University of Porto
Mushtaq Raza, University of Porto
Pedro Castro Henriques, Strongstep – Innovation in Software Quality
César Duarte, Strongstep – Innovation in Software Quality
Elias Fallon, Cadence Design Systems, Inc.
Lee Gazlay, Cadence Design Systems, Inc.
Shigeru Kusakabe, Kyushu University
Yoichi Omori, Kyushu University
Keijiro Araki, Kyushu University
Fernanda Grazioli, Universidad de la República
Silvana Moreno, Universidad de la República

November 2012

SPECIAL REPORT
CMU/SEI-2012-SR-015

Software Engineering Process Management Program

http://www.sei.cmu.edu

Carnegie Mellon

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 2012		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
TSP Symposium 2012 Proceedings	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,Software Engineering Institute,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The 2012 TSP Symposium was organized by the Software Engineering Institute (SEI) and took place September 18?20 in St. Petersburg, FL. The goal of the TSP Symposium is to bring together practitioners and academics who share a common passion to change the world of software engineering for the better through disciplined practice. The conference theme was ?Delivering Agility with Discipline.? In keeping with that theme, the community contributed a variety of technical papers describing their experiences and research using the Personal Software Process (PSP) and Team Software Process (TSP). This report contains the six papers selected by the TSP Symposium Technical Program Committee. The topics include analysis of performance data from PSP, project performance outcomes in developing design systems, and extending the PSP to evaluate the effectiveness of formal methods.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**100**	

This material is based upon work funded and supported by the United States Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the

SEI Administrative Agent
ESC/CAA
20 Schilling Circle, Building 1305, 3rd Floor
Hanscom AFB, MA 01731-2125

Table of Contents

List of Figures

List of Tables

Abstract

The 2012 TSP Symposium was organized by the Software Engineering Institute (SEI) and took place September 18–20 in St. Petersburg, FL. The goal of the TSP Symposium is to bring together practitioners and academics who share a common passion to change the world of software engineering for the better through disciplined practice. The conference theme was "Delivering Agility with Discipline." In keeping with that theme, the community contributed a variety of technical papers describing their experiences and research using the Personal Software Process (PSP) and Team Software Process (TSP). This report contains the six papers selected by the TSP Symposium Technical Program Committee. The topics include analysis of performance data from PSP, project performance outcomes in developing design systems, and extending the PSP to evaluate the effectiveness of formal methods.

1 Introduction

William Nichols, Software Engineering Institute

The 2012 TSP Symposium was organized by the Software Engineering Institute (SEI) and took place September 18–20 in St. Petersburg, FL. The goal of the TSP Symposium is to bring together practitioners and academics who share a common passion to change the world of software engineering for the better through disciplined practice. The conference theme was "Delivering Agility with Discipline." In keeping with that theme, the community contributed a variety of technical papers describing their experiences and research using the Personal Software Process (PSP) and Team Software Process (TSP).

The technical program committee was geographically and technically diverse and included members from the United States, Mexico, Portugal, South Africa, and Japan. Representatives from the SEI, academic partners, and TSP Transition Strategic Partners were included on the committee, which reviewed and selected presentations for the symposium and papers for this report.

This report contains six papers that dive deeper into experience and analysis than is feasible in a conference presentation. Topics include analysis of PSP process improvement data, integration of formal methods into the PSP for Engineers, improved analysis tools for use in the PSP for Engineers, and a TSP case study for a large company enhancing legacy products. An overview of these papers follows.

Analysis of Code Defect Injection and Removal in PSP (Diego Vallespir and William Nichols) extends research reported last year to include analysis of defects injected during the Code phase. Defects injected in the Code and Design phases are categorized by defect type. The find-and-fix times for each type are analyzed by phase removed. It is hardly surprising to PSP instructors that the find-and-fix times vary by phase. How the effort for defect type categories differs is more or less consistent with our experience, yet is described formally for the first time in this paper.

PSP_{DC} – An Adaptation of the PSP to Use Design by Contract (Silvana Moreno, Álvaro Tasistro, and Diego Vallespir) investigates using the PSP to measure the effectiveness of "Design by Contract." Design by Contract was pioneered by Bertrand Myers and enhances the design process with more formal methods. This paper includes a discussion of how the PSP process can be adapted for use in Design by Contract environments, including through the use of a verifying compiler.

Analysis of Product Quality Improvement with PSP (Fernanda Grazioli and William Nichols) analyzes and compares some of the results from two versions of the PSP introduction course, the eight-exercise PSP I/PSP II and the seven-exercise PSP Fundamentals/PSP Advanced. The performance as measured by defect level in the unit test phase is described for each exercise and compared by PSP exercise and PSP process level. The results are then analyzed using ANOVA to attempt to separate the effects of PSP level from that of progression through the exercises. Both courses showed significant improvements in quality, but the rapid pace of introduction leaves the

causality not fully inferable from this ANOVA. The significance and scale of improvement remain consistent across all versions of the PSP training courses.

Model and Tool for Analyzing Time Estimation Performance in PSP (César Duarte, João Pascoal Faria, Mushtaq Raza, and Pedro Castro Henriques) reports on research toward automated data collection and analysis tools for use in the PSP training course. The approach is promising for identifying performance issues and improving the quality, consistency, and cost of training.

Case Study: Applying and Introducing TSP to the Electronic Design Automation Industry at Cadence Design Systems (Elias Fallon and Lee Gazlay) provides an experience report from the initial stages of a large company developing software for the design of semiconductor circuits. The product includes a large legacy component. The authors describe early planning and quality results and their search for meaningful and useful size proxies in this environment.

A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report (Shigeru Kusakabe, Yoichi Omori and Keijiro Araki) is another paper describing the integration of formal methods into PSP development. The researchers propose integrating the Vienna Development Method (VDM) into the PSP for Engineers. The objective feedback from process data would help developers realize the benefits of applying more formal techniques. Initial results are promising, indicating not only reductions in defects but also more developer awareness of the results. Moreover, developers reported that they would be unlikely to implement a process improvement plan without the PSP structure.

2 An Analysis of Code Defect Injection and Removal in PSP

Diego Vallespir, Universidad de la República
William Nichols

2.1 Introduction

A primary goal of software process improvement is to make software development more effective and efficient. Because defects require rework, one path to performance improvement is to quantitatively understand the role of defects in the process. We can then make informed decisions about preventing defect injection or committing the effort necessary to remove the injected defects. The PSP establishes a highly instrumented development process that includes a rigorous measurement framework for effort and defects. After examining a large amount of data generated during PSP classes, we can describe how many defects are injected during the PSP Code phase, the types of defects injected, when they are detected, and the effort required to remove them. We have found that even using a rigorous PSP development process, nearly a quarter of all defects injected will escape into unit test. Moreover, finding and removing defects in unit test required seven times as much effort as removing them in earlier phases. The purpose of this study is not to measure the effectiveness of PSP training, but rather to characterize the defects developers inject and must subsequently remove. By examining the characteristics of defect injections and escapes, we might teach developers how to define and improve their own processes and thus make the product development more effective and efficient.

Watts Humphrey describes PSP as "a self-improvement process that helps you to control, manage, and improve the way you work" [Humphrey 2005]. This process includes phases that you complete while building the software. For each phase, the engineer collects data on the time spent in the development phase and data about the defects injected and removed.

During the PSP course, the engineers build programs while they progressively learn the PSP. We analyzed eight exercises from this PSP course version. In this section, we present an analysis of defects injected during the Code phase of the last three PSP programs (6, 7, and 8). The engineers used the complete PSP when they built these programs.

We focused on defects injected during the Code phase because these specific data had not been studied before. Recently we made a similar analysis but focused on the defects injected during the Design phase of PSP [Vallespir 2011]. Previous studies did not have all the defect data, such as defect types and individual defect times; they had only summaries.

Our analysis of the complete data available from individual defect logs shows not only that the defects injected during Code are more expensive to remove in Test than in previous phases of the process, but also that they are easy to remove in the Code Review phase. The difference is striking: it costs seven times more to remove a defect in the PSP Unit Test than it does to remove a defect during code review.

To show this, we observed how defects injected during Code escaped into each subsequent phase of the PSP and how the cost to remove them was affected by defect type and phase. We describe the different defect types injected during Code and how these defect types compare with respect

to the find-and-fix time. From this analysis, we show that "syntax" type of defects are the most common Code phase defect (around 40% of all the defects), that personal code review is an effective removal activity, and that finding and fixing Code defects in the Code Review phase is substantially less expensive than removing them in the Test phase.

Other studies have examined software quality improvement using PSP [Paulk 2006, 2010; Wohlin 1998; Rombach 2008; Hayes 1997; Ferguson 1997]. In the context of PSP, quality is measured as defect density (defects/KLOC). Our study differs from the other studies in that we focused on code defects, considered the defect type, and did not consider defect density. Instead, we concentrated on the characteristics of the defects introduced in Code. Our findings resulted from analyses of the defect types injected, how they proceeded through the process until they were found and removed, and the cost of removal in subsequent development phases. In the literature, we do not know of any other study that has the characteristics of our research.

2.2 The Personal Software Process and the Collection of Data

For each software development phase, the PSP has scripts that help the software engineer follow the process correctly. The phases include Planning, Detailed Design, Detailed Design Review, Code, Code Review, Compile, Unit Test, and Post Mortem. For each phase, the engineer collects data on the time spent in the phase and the defects injected and removed. The defect data include the defect type, the time to find and fix the defect, the phase in which the defect was injected, and the phase in which it was removed. Figure 1 shows the guidance, phases, and data collection used with the PSP.

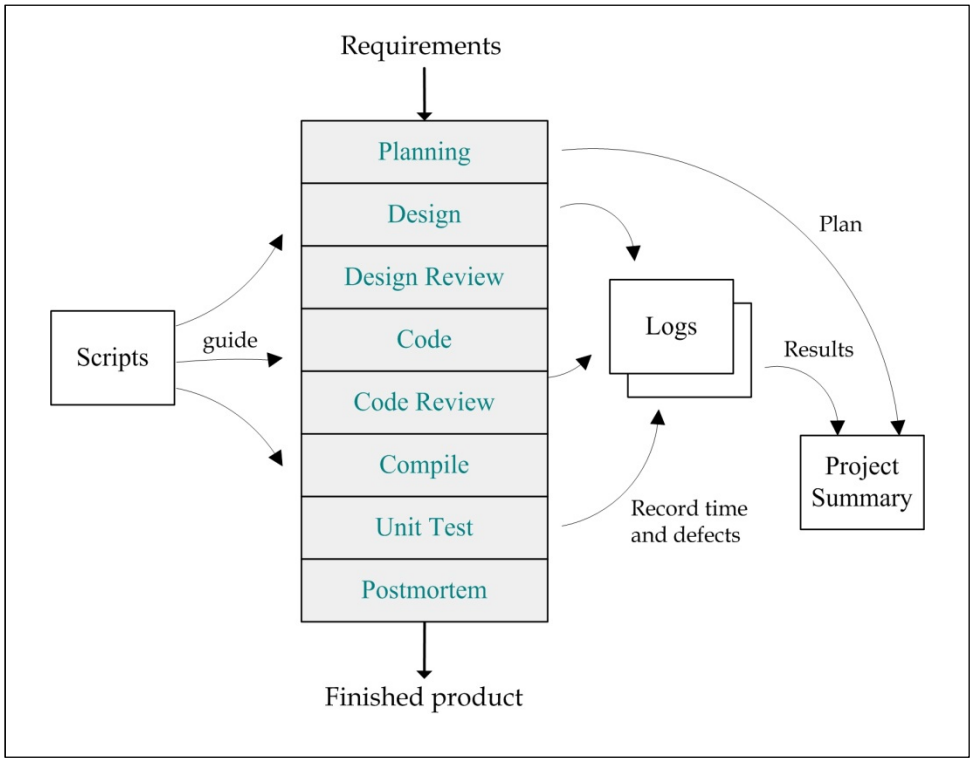


Figure 1: The PSP Phases, Scripts, Logs, and Project Summary

Some clarifications are needed to understand the measurement framework. The phases should not be confused with the activity being performed. Students are asked to write substantial amounts of code, on the scale of a small module, before proceeding through to reviewing, compiling, and testing. Once a block of code has passed into a phase, all time is logged in that phase, regardless of the developer activity. For example, a failure in test will require some coding and a compile, but the time will be logged in the "Unit Test" phase. If a personal review is performed prior to compiling, the compile can serve as an independent check of review effectiveness. We expect the compiler to remove the simple and inexpensive defects; however, if the review was effective the compile should be clean. When a defect is found, data recorded includes the phase of removal, the direct effort required to find and remove that defect, the phase in which it was injected, and the defect type and description.

The PSP defines 10 types of defects to be used during the course [Humphrey 2005; Chillarege 1996]. Table 1 presents these types of defects together with a brief description of which defects should be registered for each type.

Table 1: Defect Types in PSP

Defect Type	Possible Defects for the Type
Documentation	Comments, messages
Syntax	Spelling, punctuation, typos, instruction formats
Build/Package	Change management, library, version control
Assignment	Declaration, duplicate names, scope, limits
Interface	Procedure calls and references, I/O, user formats
Checking	Error messages, inadequate checks
Data	Structure, content
Function	Logic, pointers, loops, recursion, computation, function defects
System	Configuration, timing, memory
Environment	Design, compile, test, or other support system problems

The time to find and fix a defect is a direct measure of the time it takes to find it, correct it, and then verify that the correction made is right. In the Design Review and Code Review phases the time to find a defect is essentially zero, since finding a defect is direct in a review. However, the time to correct it and check that the correction is right depends on how complex the correction is. These variable costs are distinct from the predictable cost of performing a review. The point is that the variable cost of defects found in review can be directly measured and compared to similar costs in unit test.

On the other hand, both in the Compile and the Unit Test phases, finding a defect is an indirect activity. First, there will be a compilation error or a test case that fails. If that failure is taken as a starting point (compilation or test), what causes it (the defect) must be found in order to make the correction and verify if it is right.

During the PSP course, the engineers build programs while progressively learning PSP planning, development, and process assessment practices. For the first exercise, the engineer starts with a simple, defined process (the baseline process, called PSP 0); as the class progresses, new process steps and elements are added, from Estimation and Planning to Code Reviews, Design, and Design Review. As these elements are added, the process changes. The name of each process and which elements are added in each one are presented in Figure 2. The PSP 2.1 is the complete PSP process.

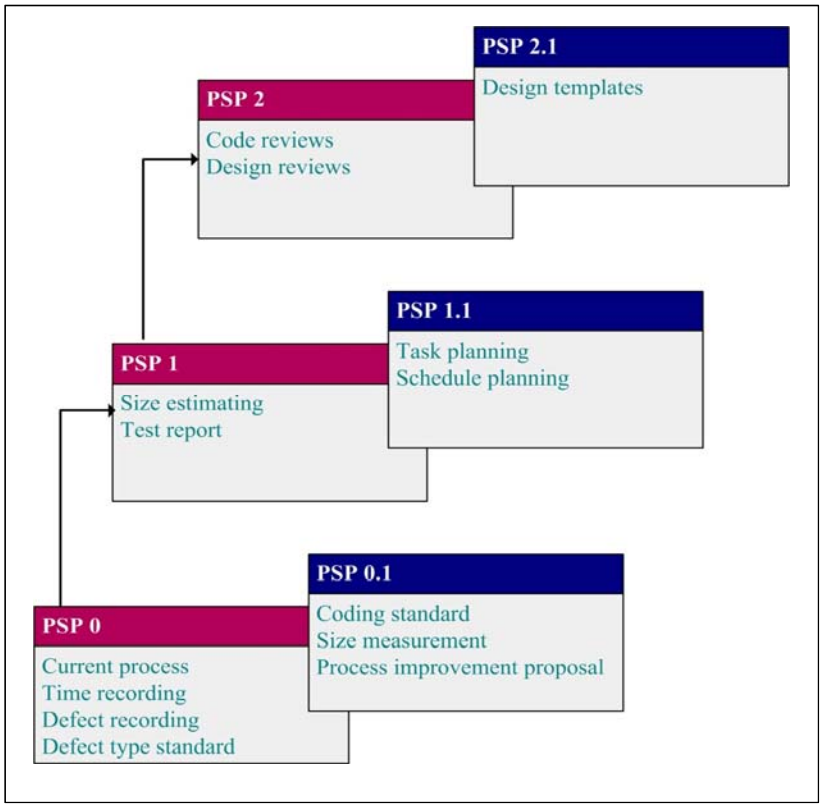


Figure 2: PSP Process Level Introduction During Course

In this section, we present an analysis of defects injected during the Code phase of the PSP, in programs 6, 7, and 8 (all developed using PSP 2.1, the complete PSP). In PSP 2.1, students conceptualize program designs prior to coding and record the design decisions using functional, logical, operational, and state templates. Students then perform a checklist-based personal review of the design to identify and remove design defects before beginning to write code. After coding, students perform a checklist-based personal review of the code. After the review they compile the code, and finally they make unit testing.

2.3 The Data Set

We used data from the eight-program version of the PSP course (PSP for Engineers I and II) taught between October 2005 and January 2010. These courses were taught by the SEI at Carnegie Mellon University or by SEI partners, including a number of different instructors in multiple countries.

This study is limited to considering only the final three programs of the PSP course (programs 6, 7, and 8). In these programs, the students apply the complete PSP process, using all process elements and techniques. Of course, not all of these techniques are necessarily applied well because the students are in a learning process.

This relative inexperience of the students constitutes a threat to the validity of this study, in the sense that different (and possibly better) results can be expected when the engineer continues using PSP in his or her working environment after taking the course. This is due to the fact that

the engineer continues to assimilate the techniques and the elements of the process after having finished learning the PSP.

We began with the 133 students who completed all programming exercises of the courses mentioned above. From this we made several cuts to remove errors and questionable data and to select the data most likely to have comparable design and coding characteristics.

Because of data errors, we removed data from three students. Johnson and Disney reviewed the quality of the PSP data [Johnson 1999]. Their analysis showed that 5% of the data were incorrect; however, many or most of those errors in their data were due to process calculations the students made. Process calculations are calculations made to obtain the values of certain derived measures the process uses to make estimates for the next program, such as the defects injected per hour in a certain phase or the alpha and beta parameters for a linear regression that relates the estimated size to the real size of the program.

Because our data were collected with direct entry into a Microsoft Access tool, which then performed all process calculations automatically, the amount of data removed (2.3%) is lower than the percentage reported by Johnson and Disney; however, this amount seems reasonable.

We next reduced the data set to separate programming languages with relatively common design and coding characteristics. As we analyze the code defects, it seems reasonable to consider only languages with similar characteristics that might affect code size, modularity, subroutine interfacing, and module logic. The students used a number of different program languages, as shown in Figure 3.

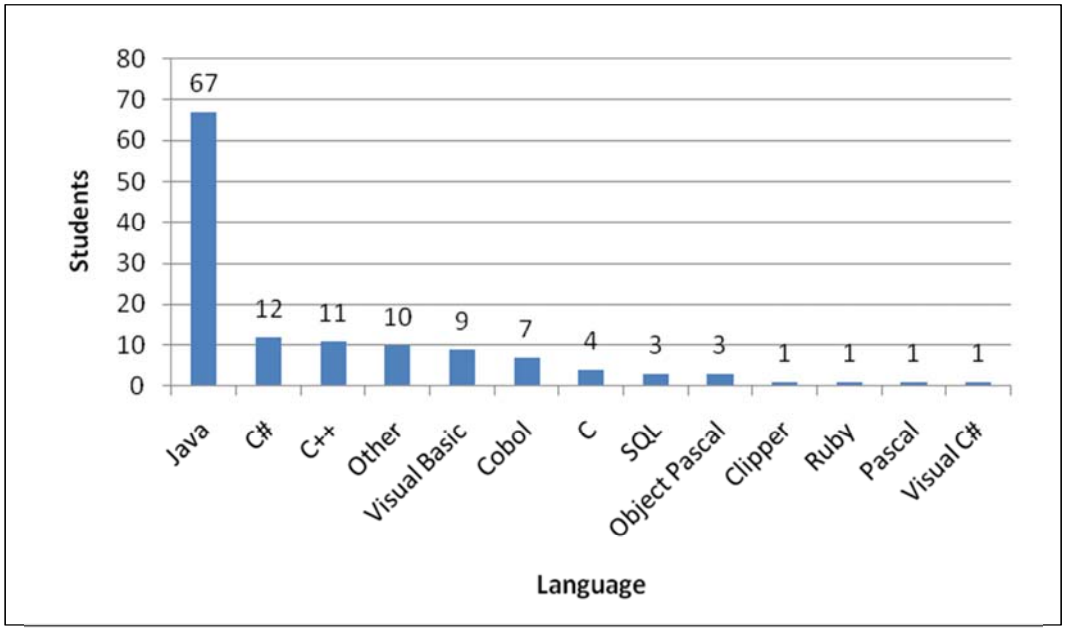


Figure 3: Quantity of Students by Program Languages

The most common language used was Java. To increase the size of the data set, we decided to include the data generated by students who used Java, C#, C++, and C. The languages in this group use similar syntax, subprogram, and data constructs. For the simple programs produced in the PSP course, we judged that these were most likely to have similar modularization, interface, and data design considerations. This cut reduced our data to 94 subjects.

Because our intent was to analyze defects, we removed from consideration any data for which defects were not recorded. From the 94 engineers remaining, two recorded no defects at all in the three programs considered. Our data set for this analysis was, therefore, reduced to 92 engineers. In the following sections we present the types of defects that were injected in the Code phase, when the defects were removed, and the effort required to find and fix these defects.

Our analysis included the defect injection and removal performance of individuals and the performance range of variation among them. It should be noted that this is different from analyzing the behavior of a team. That is, we wanted to characterize the work of individual programmers, which is why we calculated individual performance for each one of the subjects. After obtaining the performance of each subject, we computed the mean and spread of individual averages (as opposed to the global average for all defects). We calculated an estimate of the mean percentage, the 95% confidence interval for that mean (to characterize the standard error on the mean), and the standard deviation of the distribution, to characterize the spread among individuals. For these calculations, as previously mentioned, only programs 6, 7, and 8 of the PSP course were used in order to consider the complete PSP.

For this study we included the 92 engineers who recorded defects. In several cases, the number of engineers included varies, and in each of these cases the motive for varying is documented.

2.4 Where the Defects Are Injected

The first goal of our analysis was to better understand where defects were injected. We expected injections to be dominated by the Design and Code phases, because they are the construction phases in PSP. We began by explicitly documenting the phase injection percentages that occur during the PSP course.

For each PSP phase and for each individual, we calculated the percentage of defects injected. The distribution statistics are shown in Table 2**.**

Table 2: Mean Lower, Upper Confidence Interval Values and Standard Deviation of the Percentage of Defects Injected by Phase

	DLD	DLDR	Code	CR	Comp	UT
Mean	46.4	0.4	52.4	0.3	0.03	0.5
Lower	40.8	0.2	46.7	0.0	0.00	0.2
Upper	52.0	0.7	58.1	0.7	0.09	0.9
Std. Dev.	27.2	1.7	27.4	1.8	0.30	1.8

The Design and Code phases have similar injection percentages both on average and in the spread. Their mean of the percentage of defects injected is near 50% with lower and upper CI bounds between 40% and 58%. Both standard deviations are around 27%. So, in the average of this population, roughly half of the defects were injected in the Design phase and the other half were injected in the Code phase. On average, the defect potential of these phases appears to be very similar. The standard deviation shows, however, that the variability between individuals is substantial. Nonetheless, as we expected, in the average almost 99% of the defects were injected in the Design and Code phases with only around 1% of the defects injected in the other phases.

The Design Review, Code Review, Compile, and Unit Test phases also have similar average defect potentials. The average in all these cases is less than 0.5% and their standard deviations are small, the largest being 1.8% in Code Review and Unit Testing. This shows that during verification activities in PSP, the percentage of defects injected is low but not zero. From time to time, developers inject defects while correcting other defects. We will study these secondary injections in a later study.

The variability between individuals and the similarity between the Code and Design phase is also presented in Figure 4. Note that the range in both phases is from 0% to 100% (all possible values). The 25th percentile is 26.34 for Design and 35.78 for Code, the median is 45.80 for Design and 52.08 for Code, and the 75th percentile is 64.22 for Design and 71.56 for Code.

Despite a high variability between individuals, this analysis shows that the great majority of defects are injected in the Design and Code phases. Slightly more defects are injected during Code than during Design, but the difference is not statistically significant. We could, therefore, focus on the defects injected in the Design and Code phases. In this article, we discuss only the defects injected in the Code phase.

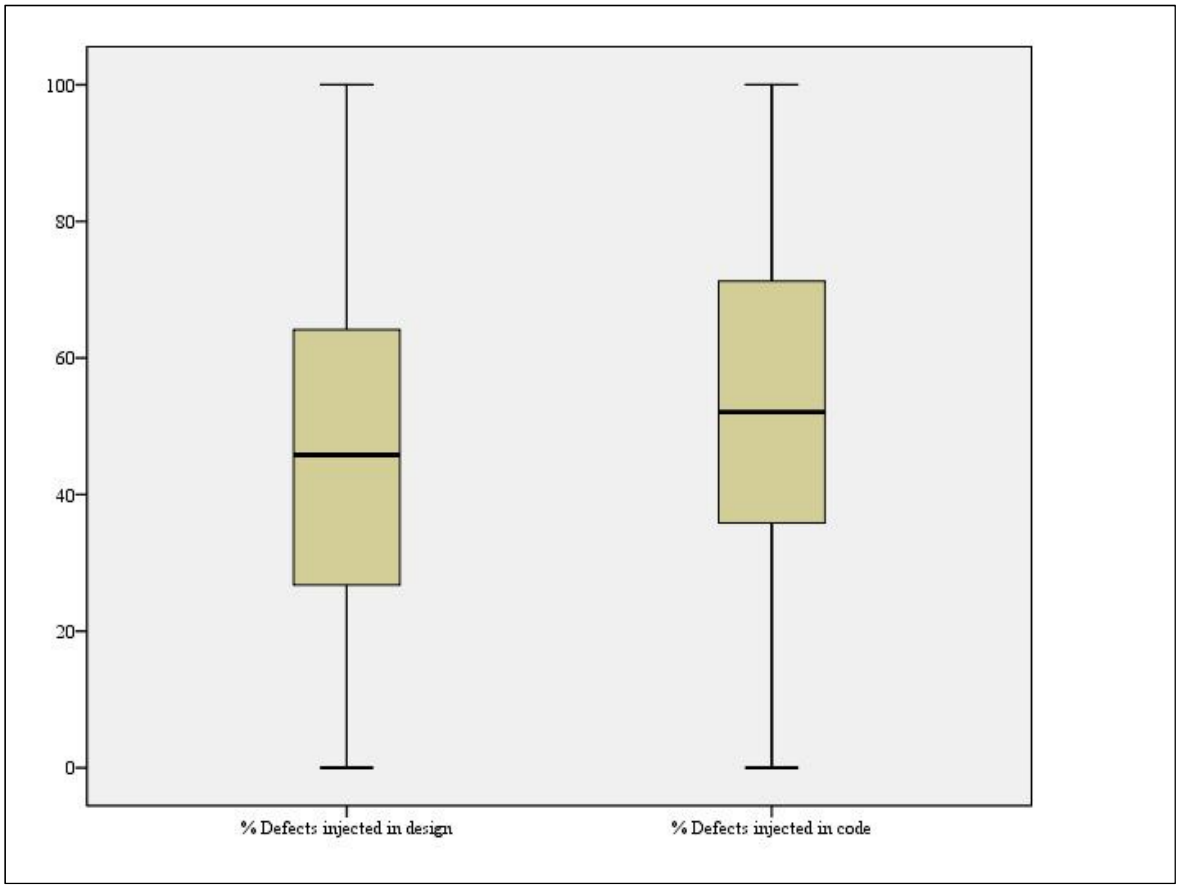


Figure 4: Percentage of Defects Injected by Phase (Box and Whisker Chart)

2.5 Analysis of CODE Defects

From the 92 engineers in our data set there were four whose records of injected defects (injected during Code) were uncertain regarding their correctness; they were therefore dismissed for this analysis. Also, eight engineers did not record defects in the Code phase, so they were dismissed, as well.

Our data set for analysis of the code defects was, therefore, reduced to 80 engineers. In the following sections we discuss the types of defects that are injected in the Code phase, when those defects are removed, and the effort required to find and fix the defects.

2.5.1 Defect Types Injected During the Code Phase

To improve the detection of code defects, we first wanted to know which types of defects were injected during the Code phase. Table 3 shows the mean of the percentage of the different defect types injected. It also presents the lower and upper bound of the 95% confidence interval for the mean (a measure of the standard error) and the standard deviation of the distribution.

None of the engineers registered system type defects injected during the Code phase. The Mean D. line presents what was found in our previous work of analysis of the defects injected during the Design phase, so these results could be comparable.

Table 3: Percentage of Defect Types Injected During Code

	Documentation	Syntax	Build Package	Assignment	Interface	Checking	Data	Function	System	Environment
Mean	3.8	40.3	0.6	14.0	5.5	2.7	5.8	26.4	0	0.9
Mean D.	6.9	6.0	0.1	12.6	10.0	4.6	9.8	46.6	0.2	3.1
Lower	1.5	33.7	0.0	9.9	3.1	1.0	3.1	19.9	0	0.0
Upper	6.0	46.9	1.1	18.1	8.0	4.4	8.6	32.8	0	1.7
Std. Dev.	10.1	29.5	2.5	18.4	11.1	7.4	12.4	29.1	0	3.9

When seeking improvement opportunities, a Pareto sort can be used to identify the most frequent or most expensive types. These types can then be the focus for future improvement efforts. For the following analysis we sorted defects by frequency, and then segmented the defect types into three categories of increasing frequency. The first grouping is “very few” defects of this type. In the “very few” category we found system, build/package, and environment types of defects. In our previous work, in which we studied the defects injected in the Design phase, we found within this category the system and build/package defect types, but not the environment type of defect. Considering this work and the previous work, it is clear that, during the PSP course, the build/package and system types of defects were seldom injected. This may be due to the PSP course exercises rather than the PSP. Because the exercises are small, take only a few hours, contain few components, and make few external library references, build packages are usually

quite simple. We expect to find more defects of these types in the TSP [Humphrey 2000, 2006] in industrial-scale projects.

A second grouping is “few defects”; most of the other defect types (all except syntax and function types) are in this category. The percentage of defects in this category ranged from 2.7% to 14.0%. In our previous work, both syntax and environment defect types were in this category. It is reasonable that, when analyzing the design defects mentioned, we find few syntax defects and that the percentage of these defects in relation to the rest of the other types of defects increases when the defects injected during the Code phase are analyzed. It is natural that when coding, more syntax defects are made than when designing, even if the design contains pseudocode, as in the case of the PSP.

The third and final grouping, “many defects,” includes the syntax and function types of defects. The syntax defects injected during Code are around 40% of the total defects and the function defects are around 26%. Approximately two out of three injected defects during the Code phase are of one of these two types.

As mentioned earlier, one out of four (26.4%) defects injected during the Code phase is a function type of defect. This is an opportunity for improvement for the PSP, since this type of defect should be injected (and as far as possible removed) before reaching Code phase. The fact that there is such a large percentage of this type of defect injected indicates problems in the Design and Design Review phases. PSP incorporates a detailed pseudocode in the Design phase using the Logic Template. Therefore, it is in this phase that the function type defects should be injected, and then they should be removed in the Design Review phase.

The lower, upper, and standard deviation data show again the high variability between individuals. This can also be observed in Figure 5; the box and whisker chart shows many observations as outliers. The high variability among individuals is repeated in every analysis conducted, both in this work and in the previous work, which studied the defects injected in the Design phase. A detailed analysis of developer variability is beyond the scope of this paper.

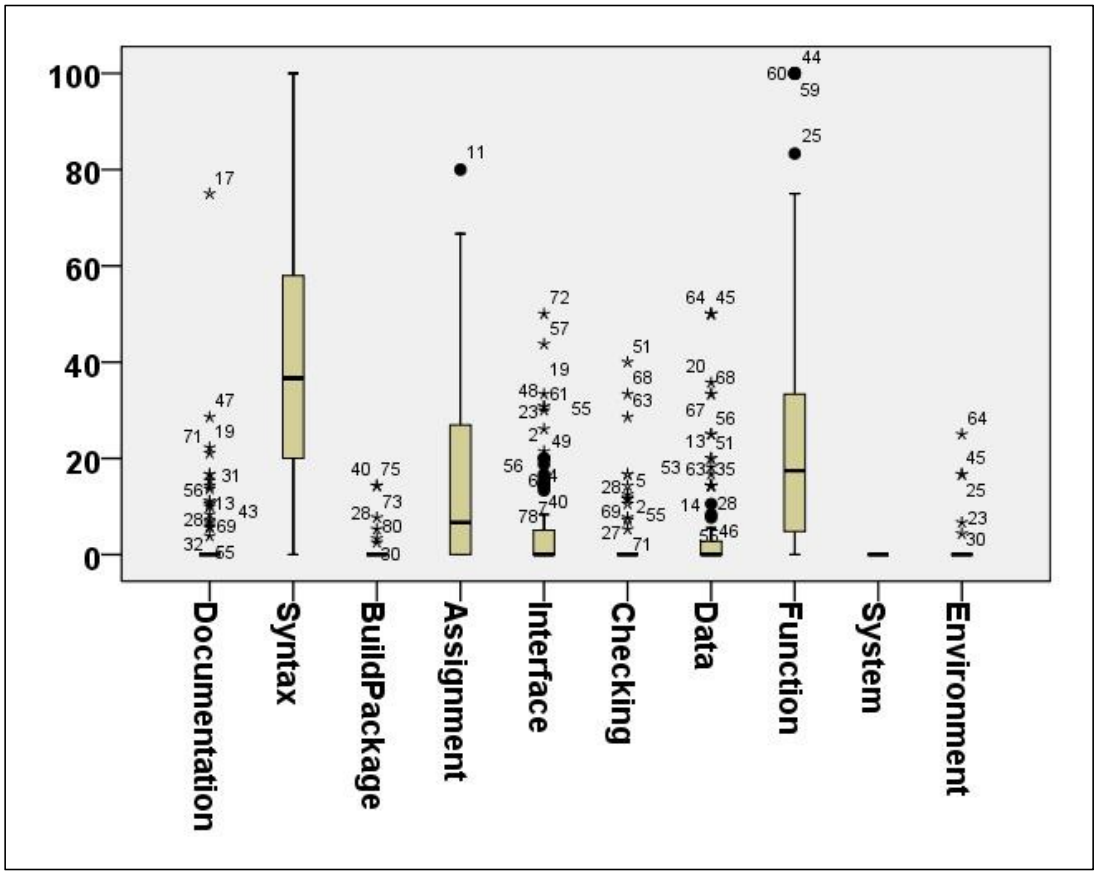


Figure 5: Box and Whisker Plot of the Percentage of Defects Injected During Code

2.5.2 When Are the Defects Injected During Code Removed?

Our analysis indicated the subsequent phases during which the code defects were removed. While our data set was large, it remained too small for us to examine the removals based on defect type. Still, for each engineer who injected defects in the Code phase, we identified the phases in which the engineer found the defects; then, for every phase, we determined the percentage of the defects that were found in that phase.

Table 4 shows the mean (with 95% confidence interval) and standard deviation for the different phases. As previously shown, the standard deviation was high, indicating the high variability between individuals. From this we learned that, on average, 62% of the defects injected during Code were found in the Code Review phase.

Table 4: Phases Where the Code Defects are Found (Percentage)

	CODE DEFECTS		
	CR	**Comp**	**UT**
Mean	62.0	16.6	21.4
Lower	55.0	11.7	15.4
Upper	69.0	21.6	27.3
Standard Deviation	31.3	22.4	26.9

In our previous analysis, we found that around 50% of the defects injected during the Design phase were detected in the Design Review phase. This indicates that for the defects injected in both the Design and Code phases, the following Review phases are highly effective.

On the other hand, in the previous study we also found that around 25% of the defects injected in the Design phase are detected only in Unit Test. This happens in 21.4% of the cases, based on our analysis of defects injected in the Code phase. This indicates that a relatively high percentage of defects manage to escape from the different detection phases and reach Unit Test.

We also know, of course, that not all the defects that escape into Unit Test are found in Unit Test. This phase will not have a 100% yield. (That is, we will not find all the defects that are in a given unit when it arrives at Unit Test.) Therefore the percentage of defects found in each of these phases is smaller than reported, while the actual percentage of escapes into Unit Test is a lower limit. An estimate or measurement of the Unit Test yield will be necessary to revise these phase estimates.

Figure 6 shows the box and whisker charts displaying the percentage of defects found in the different phases. Figure 6also shows the high variability between individuals in the percentage of defects found during Code Review, Compile, and Unit Test phases. This variability among individuals was also found in the previous study.

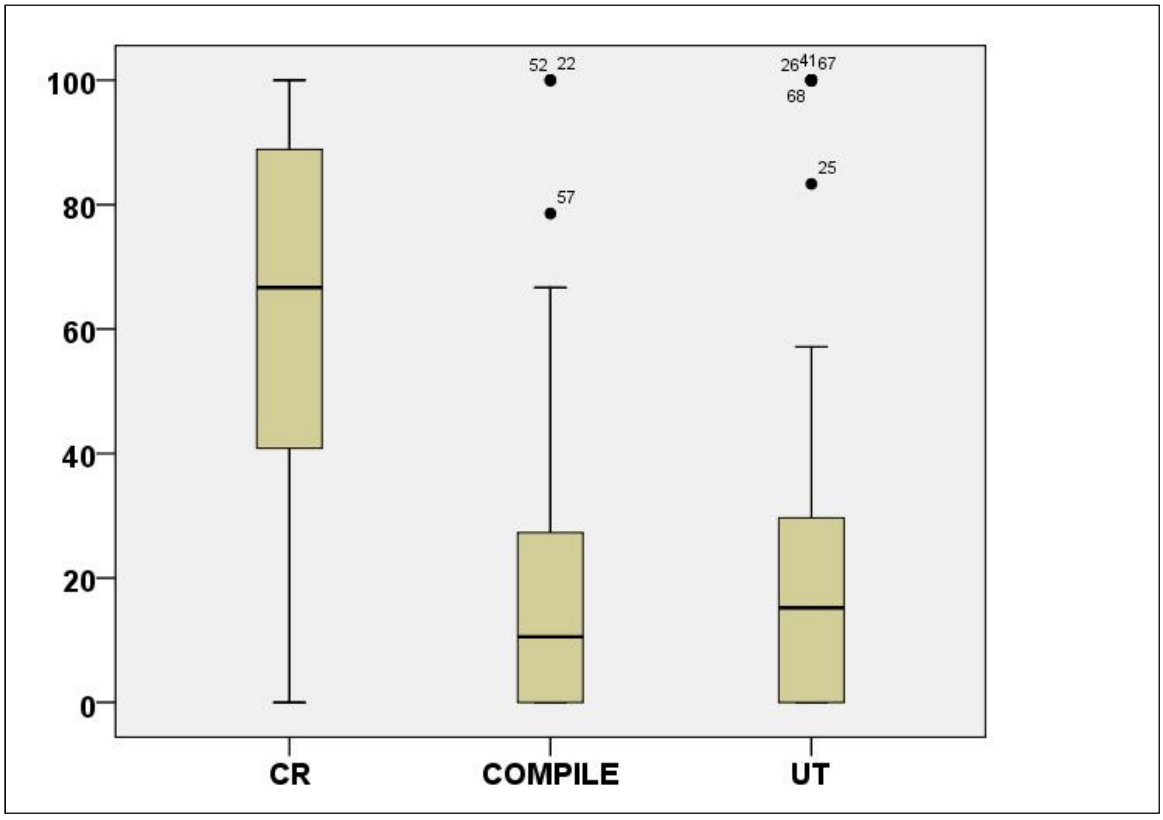


Figure 6: In Which Phase Are the Code Defects Found? – Variability Between Individuals

2.5.3 Cost to Remove the Defects Injected in Code

What is the cost and variation in cost (in minutes) to find and fix the defects that are injected during Code? To determine this, we first analyze the differences in cost, segmented by the removal phase. Second, we study the differences in cost segmented by defect type.

It also would be interesting to segment and analyze both the removal phase and the defect type jointly. Unfortunately, because of limited sample size after a two-dimensional segmentation, we cannot perform that analysis with statistical significance.

2.5.3.1 Phase Removal Cost

What is the cost, in each removal phase, to find and fix a defect injected in Code? Code defects can be removed in PSP in the Code Review (CR), Compile, and Unit Test phases. For each engineer, we calculated the average task time of removing a design defect in each of the different phases. Because some engineers did not remove code defects in one or more phases, our sample size varied by phase. We had data from 72 engineers for Code Review, 44 for Compile, and 51 for Unit Test.

Table 5 shows the mean, lower, and upper 95% confidence intervals and the standard deviation for the find-and-fix time (in minutes) for code defects in each of the studied phases.

Table 5: Cost of Find and Fix Defects Injected in Design Segmented by Phase Removed

	CODE DEFECTS		
	CR	**Com**	**UT**
Mean	1.9	1.5	14.4
Lower	1.5	1.1	9.8
Upper	2.3	1.9	19.0
Standard Deviation	1.9	1.3	16.4

As we expected, the average cost of finding code defects during Unit Test is much higher than in the other phases, by a factor of seven. We are not stating here that the cost of finding and fixing a particular code defect during Unit Test is seven times higher than finding and fixing the same particular code defect in Code Review or Compile. We are stating that with the use of PSP, the code defects that are removed during Unit Test cost seven times more than the ones that were removed in Code Review and Compile (these are different defects).

In our previous study we also found that the Design injection defects find-and-fix times in Design Review and Code Review are a factor of five smaller than the find-and-fix times in Unit Test. We also found that the defects injected in Design and removed in Unit Test have an average find-and-fix time of 23 minutes. Considering the two analyses, these are the defects that are most costly to remove. Defects injected in Code and removed in Unit Test follow with an average of 14.4 minutes. Testing, even at the unit test level, is consistently a more expensive defect removal activity than alternative verification activities.

We also found high variability among individual engineers. This variability can be seen in the box and whisker chart in Figure 7. We tested for normal distribution after log transformation of find-and-fix times for Code Review, Compile, and Unit Test. Only Unit Test is consistent ($p > 0.05$

using Shapiro-Wilk test) with a log-normal distribution. The test for normality is primarily useful to verify that we can apply regression analysis to the transformed data; however, understanding the distribution also helped to characterize the asymmetry and long-tailed nature of the variation. The log-normality confirmed our intuition that some defects found in Unit Test required far more than the average effort to fix, making Unit Test times highly variable. We observe that both the mean and variation of rework cost in the Code Review and Compile phases were significantly lower than Unit Test in both the statistical sense and the practical sense.

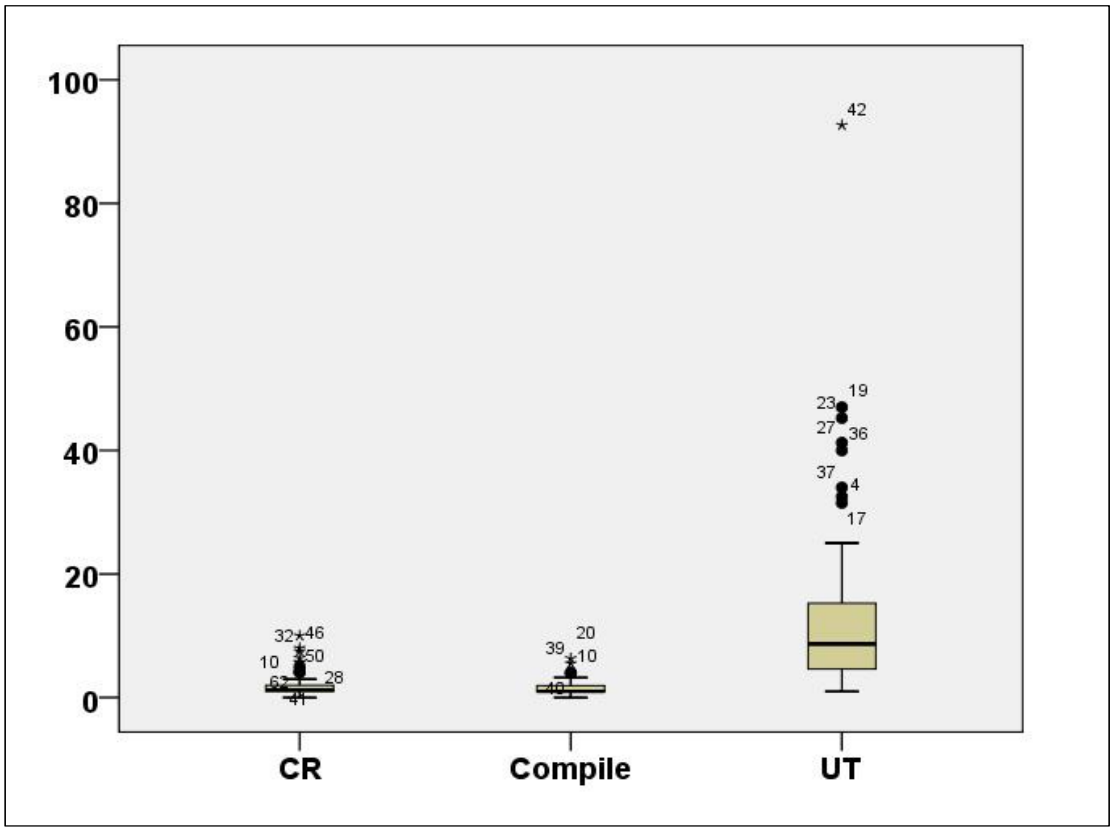


Figure 7: Box and Whisker Plot of the Cost to Find and Fix a Code Defect Segmented by Phase Removed

2.5.3.2 Defect Removal by Type

What is the find-and-fix cost, per defect type, of defects injected during Code? We did not have enough data for a statistically significant analysis of the cost of removing the build/package, checking, system, and environment types of defects. However, we were able to analyze the remaining defect types.

Table 6 presents the mean, lower, and upper 95% confidence interval and the standard deviation categorized by defect type then sorted by the find-and-fix cost of code defects. For prioritization, the cost, in minutes, for "find-and-fix" fell into three groups:

1. A group that has a mean around 2-3 minutes: documentation, syntax, assignment, and interface defects
2. A group, composed only of function defects, that has a mean around 9 minutes

3. A group, composed only of data defects, that has a mean around 12 minutes

Function type defects (injected during Code phase) take three times longer to find and fix than documentation, syntax, assignment, and interface defects. Data type defects take four times as much time.

Table 6: Cost of Find-and-Fix Defects Injected in Code Segmented by Defect Type

	Documentation	Syntax	Assignment	Interface	Data	Function
Mean	3.4	1.9	2.7	2.3	12.2	9.4
Lower	1.3	1.4	1.8	1.4	0.0	6.8
Upper	5.4	2.3	3.7	3.3	27.2	12.1
Standard Deviation	4.2	2.0	3.1	2.2	32.9	10.7

As in the other cases, the variation among individual developers was high. This can be seen using the standard deviation, as well as the box and whisker chart that is presented in Figure 8.

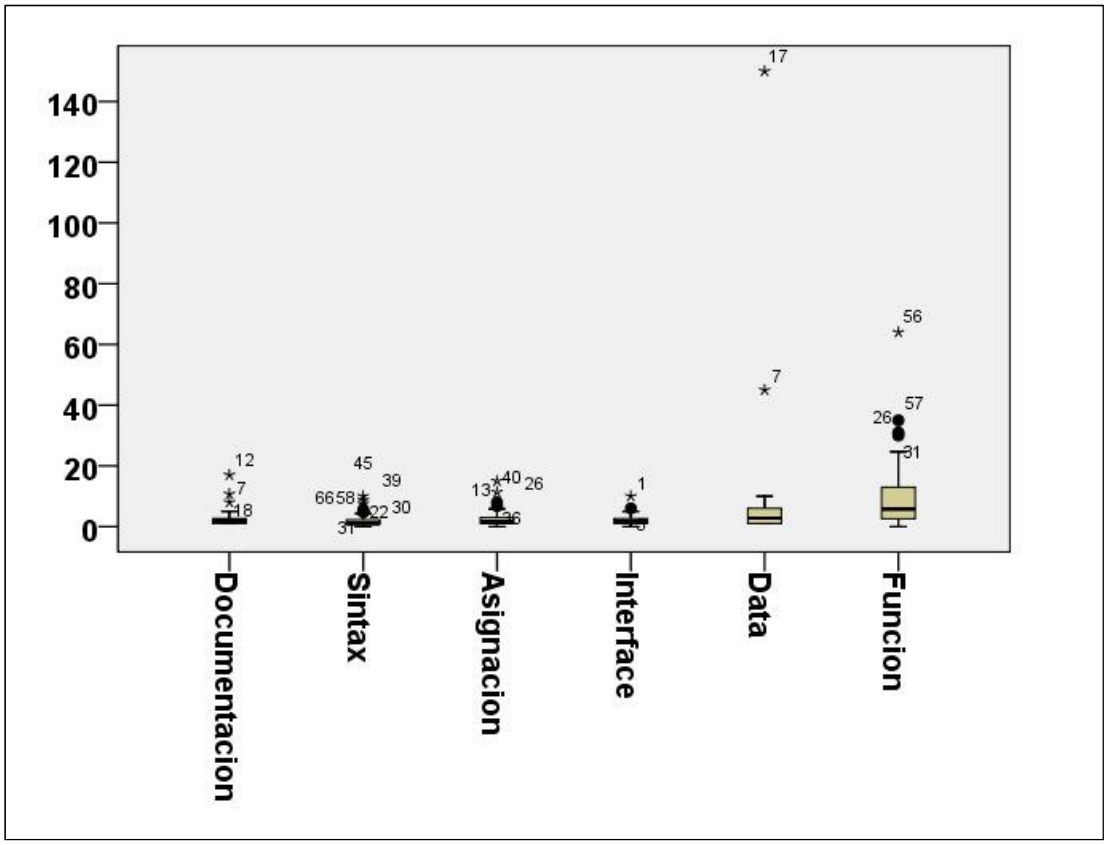


Figure 8: Box and Whisker Plot of the Cost to Find and Fix a Defect Segmented by Defect Type

2.6 Limitations of this Work

There are several considerations that limit the ability to generalize this work: the limited life cycle of the PSP course, the lack of a production environment, that students are still learning process, the students perform the defect categorization, and the nature of the exercises. Because the PSP life cycle begins in Detailed Design and ends in Unit Test, we do not observe all types of defects and specifically do not observe requirements defects or those that would be found in the late testing such as Integration Test, System Test, and Acceptance Test. This also implies that finds in Unit Test are only a lower estimate of the actual escapes into Unit Test. Defects such as build and environment or requirements injections are not considered.

The second consideration is that the PSP exercises do not build production code. Code is not intended to be "bullet proofed" or production ready. This is most likely to affect the rigor of Unit Test. Students often run only the minimum tests specified. This will likely lead to fewer defects being found and higher overall development rates. For example, coding rates are typically much higher than those found in industry. Also excluded is the production practice of peer inspections.

A third consideration is that, students using PSP are still learning techniques of design and personal review. The results after gaining experience may differ from those found during this course.

A fourth consideration is that precision of the student categorization of defect types has not been precisely measured. That is, students are learning how to categorize defects and should receive

guidance from the instructor. Nonetheless, there will certainly be some inconsistency among the students in how specific defects are categorized.

Finally, the problems, though modestly challenging, do not represent a broad range of development problems.

2.7 Conclusions and Future Work

In this analysis, we considered the work of 92 software engineers who, during PSP course work, developed programs in the Java, C, C#, or C++ programming languages. In each of our analyses, we observed a high variation in range of performance among individuals; we show this variability using standard deviation and box and whisker charts to display the median, quartiles, and range.

After considering this variation, we focused our analysis on the defects injected during Code. Our analysis showed that most common code defects (40%) are of syntax type. This type of defect is the cheapest to find and fix (1.9 minutes). The types of defects injected in Code that are most expensive to correct are the data (12.2 minutes) and function (9.4 minutes) defect types.

In addition, the analysis showed that build/package, systems, and environment defects were seldom injected in the Code phase. We interpreted this as a consequence of the small programs developed during the course, rather than as a characteristic of PSP as a development discipline.

We found that defects were injected roughly equally in the Design and Code phases; that is, around half of the defects were injected in code. 62% of the code defects were found early through appraisal during the Code Review phase. However, around 21% were discovered during Unit Test, where mean defect find-and-fix time is almost seven times greater than find-and-fix time in review.

While this analysis provided insights into the injection and removal profile of code defects with greater specificity than previously possible, a larger data set would allow us to consider more detail, such as the costs of defects discriminated by defect type in addition to removal phase. A more complete analysis may enable us to analyze improvement opportunities to achieve better process yields. In future analysis, we will examine the relationship between Design and Code activities and the defects found in the downstream phases.

2.8 Author Biographies

Diego Vallespir
Assistant Professor
School of Engineering, Universidad de la República

Diego Vallespir is an assistant professor at the Engineering School at the Universidad de la República (UdelaR), where he is also the director of the Informatics Professional Postgraduate Center and the Software Engineering Research Group. He is also a member of the Organization Committee of the Software and Systems Process Improvement Network in Uruguay (SPIN Uruguay).

Vallespir holds Engineer, Master Science, and PhD titles in Computer Science, all from UdelaR. He has had several articles published in international conferences. His main research topics are empirical software engineering, software process, and software testing.

William Nichols

Bill Nichols joined the Software Engineering Institute (SEI) in 2006 as a senior member of the technical staff and serves as a PSP instructor and TSP coach with the Team Software Process (TSP) Program. Prior to joining the SEI, Nichols led a software development team at the Bettis Laboratory near Pittsburgh, Pennsylvania, where he had been developing and maintaining nuclear engineering and scientific software for 14 years. His publication topics include the interaction patterns on software development teams, design and performance of a physics data acquisition system, analysis and results from a particle physics experiment, and algorithm development for use in neutron diffusion programs. He has a doctorate in physics from Carnegie Mellon University.

2.9 References/Bibliography

[Chillarege 1996]
Chillarege, R. "Orthogonal Defect Classification," 359–400. *Handbook of Software Reliability Engineering*. Edited by M.R. Lyu. McGraw-Hill Book Company, 1996.

[Ferguson 1997]
Ferguson, Pat; Humphrey, Watts S.; Khajenoori, Soheil; Macke, Susan; & Matvya, Annette. "Results of Applying the Personal Software Process." *Computer 30*, 5, (May 1997): 24–31.

[Hayes 1997]
Hayes, Will & Over, James. *The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers* (CMU/SEI-97-TR-001). Software Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/library/abstracts/reports/97tr001.cfm

[Humphrey 2006]
Humphrey, Watts S. *TSP: Coaching Development Teams*. Addison-Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/201731134.cfm

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Humphrey 2000]
Humphrey, Watts S. *The Team Software Process* (CMU/SEI-2001-TR-023). Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu/library/abstracts/reports/87tr023.cfm

[Johnson 1999]
Johnson, Philip M. & Disney, Anne M. "A Critical Analysis of PSP Data Quality: Results from a Case Study." *Empirical Software Engineering 4*, 4 (March 1999): 317–349.

[Paulk 2010]
Paulk, Mark C. “The Impact of Process Discipline on Personal Software Quality and Productivity.” *Software Quality Professional 12*, 2 (March 2010): 15–19.

[Paulk 2006]
Paulk, Mark C. “Factors Affecting Personal Software Quality.” *CrossTalk: The Journal of Defense Software Engineering 19*, 3 (March 2006): 9–13.

[Rombach 2008]
Rombach, Dieter; Munch, Jurgen; Ocampo, Alexis; Humphrey, Watts S.; & Burton, Dan. “Teaching Disciplined Software Development.” *The Journal of Systems and Software 81*, 5 (2008): 747–763.

[Vallespir 2011]
Vallespir, Diego & Nichols, William. “Analysis of Design Defects Injection and Removal in PSP,” 19–25. *Proceedings of the TSP Symposium 2011: A dedication to excellence*. Atlanta, GA, September 2011. Software Engineering Institute, Carnegie Mellon University, 2011.

[Wohlin 1998]
Wohlin, C. & Wesslen, A. “Understanding software defect detection in the Personal Software Process,” 49–58. *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998. IEEE, 1998.

3 Model and Tool for Analyzing Time Estimation Performance in PSP

César Duarte, Strongstep – Innovation in Software Quality
João Pascoal Faria, Faculty of Engineering, University of Porto
Mushtaq Raza, University of Porto, Portugal
Pedro Castro Henriques, Strongstep – Innovation in Software Quality

High maturity software development processes, making intensive use of metrics and quantitative methods, such as the Personal Software Process (PSP) and the Team Software Process (TSP), can generate a significant amount of data that can be periodically analyzed to identify performance problems, determine their root causes, and devise improvement actions. Currently, there are several tools that automate data collection and produce performance charts for manual analysis in the context of the PSP and TSP, but little tool support exists for automating the data analysis and the recommendation of improvement actions. Manual analysis of this performance data is problematic because of the large amount of data to analyze and the time and expertise required to do a proper analysis. Hence, we propose in this paper a performance model and a tool (named PSP PAIR) to automate the analysis of performance data produced in the context of the PSP, specifically, to identify performance problems and their root causes, and recommend improvement actions. The work presented is limited to the analysis of the time (effort) estimation performance of PSP developers, but is extensible to other performance indicators and development processes.

3.1 Introduction

Software development professionals and organizations face an increasing pressure to produce high-quality software in a timely manner. The PSP and TSP are examples of methodologies tailored to help individual developers and development teams improve their performance and produce virtually defect free software on time and budget [Davis 2003]. One of the pillars of the PSP/TSP is its measurement framework: based on four simple measures (effort, schedule, size, and defects) it supports several quantitative methods for project management, quality management, and process improvement [Pomeroy-Huff et al. 2009].

Software development processes that make intensive use of metrics and quantitative methods, such as the PSP/TSP, can generate a significant amount of data that can be analyzed periodically to identify performance problems, determine their root causes, and identify improvement actions [Burton 2006]. Currently, there are several tools that automate data collection and produce performance charts, tables, and reports for manual analysis in the context of PSP/TSP [Design Studio Team 1997; Software Process Dashboard 2011; C.S.D Laboratory 2010; Sison 2005; Shin 2007; Nasir 2005], but little support exists for automating the data analysis and the recommendation of improvement actions. There are some studies that show cause-effect relationships among performance indicators (PIs) [Kemerer 2009; Shen 2011], but no automated root cause analysis is proposed. The manual analysis of the performance data for determining root causes of performance problems and devising improvement actions has significant shortcomings: the amount of data to analyze may be overwhelming [Humphrey 2002], the effort needed to do the analyses is significant, and expert knowledge is required to do the analyses and devise the improvement actions.

Hence, the goal of our research is to develop models and tools to automate the analyses of performance data produced in the context of high maturity development processes, namely, to identify performance problems and their root causes and recommend improvement actions. For practical reasons, we limited the work presented in this paper to the automated analysis of time (effort) estimation performance in the context of the PSP, while at the same time providing a significant degree of extensibility for other PIs and development processes.

The rest of the paper is organized as follows: In Section 3.2, we present the performance model conceived for analyzing the time estimation performance of PSP developers. In Section 3.3, we present the tool developed for automatically analyzing time estimation performance based on the previously defined model. In Section 3.4, we present an evaluation based on a case study. In Section 3.5, we present a summary of main achievements and points of future work.

3.2 Performance Model for Analyzing the Time Estimation Performance

In this section we describe the performance model we conceived for enabling the automated analysis of the time estimating performance of PSP developers (i.e., the identification of performance problems, root causes, and remedial actions). An overview of the artifacts and steps involved in our approach is presented in Figure 9.

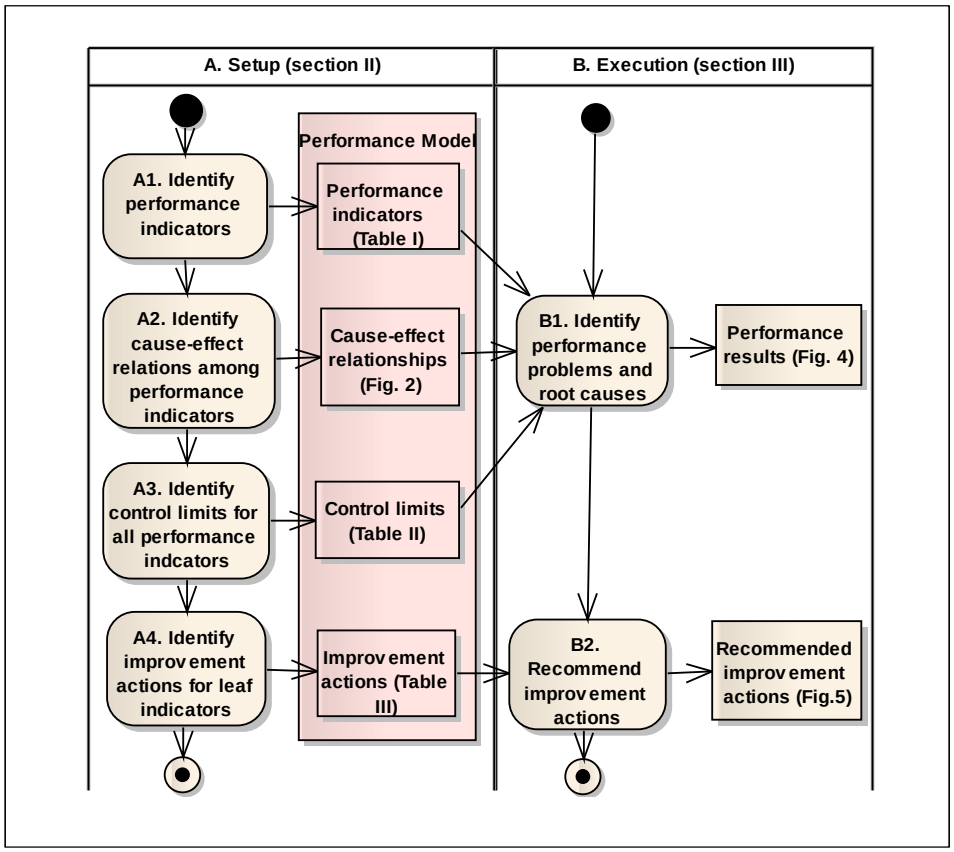


Figure 9: Overview of the Steps and Artifacts in our Approach (Notation of UML Activity Diagrams)

The performance model comprises a set of performance indicators (PI) and cause-effect relationships, for identifying root causes of performance problems (in this case, time estimation

problems); recommended control limits for the PIs, for identifying performance problems; and a list of improvement actions for each leaf PI, for identifying remedial actions.

The performance model was conceived based on PSP specifications (of base and derived measures, estimation methods, etc.); literature review, namely published analysis reports of PSP data; and opinions of PSP experts.

We focused our attention on the time estimation performance due to the following reasons: delivering products on time is one of the most important goals in every project; one of the biggest strengths that is pointed out on the PSP/TSP is the delivery of products on time (or with comparatively little deviation) and virtually defect free (as low as 0.06 defects/KLOC) [Davis 2003]. The time estimation performance is affected by the other performance aspects (namely, quality and productivity), so the analysis of time estimation performance is broad in scope. We intend in the future to build similar models for analyzing other PSP performance aspects, namely quality and productivity performance.

3.2.1 Performance Indicators and Dependencies

In the PSP, the time estimation accuracy, or time estimation error (TimeEE), is defined by the usual formula (see Table 7), and is usually presented as a percentage.

In order to identify the factors that influence the behavior of the PI, one has to look at the estimation method used in the PSP, (i.e., the PROBE method) [Humphrey, 2005]. In this method, a time (or effort) estimate is obtained based on a size estimate of the deliverable (in lines of code (LOC), function points [FPs], etc.) and a productivity estimate (in LOC/hour, FP/hour, etc.). So, the quality of the time estimate will depend on the quality of the size and productivity estimates. From the formulas of the Size Estimation Error (SizeEE) and Productivity Estimation Error (PEE) (see Table 7) and the definition of productivity as a size/time ratio, we get (see [Duarte 2012] for details):

$$TimeEE = \frac{SizeEE + 1}{PEE + 1} - 1$$

Hence, we conclude that TimeEE is affected by SizeEE and PEE according to the formula, as depicted in Figure 10. It is worth noting that, in the case of changes to existing artifacts, only the size of the changes is considered.

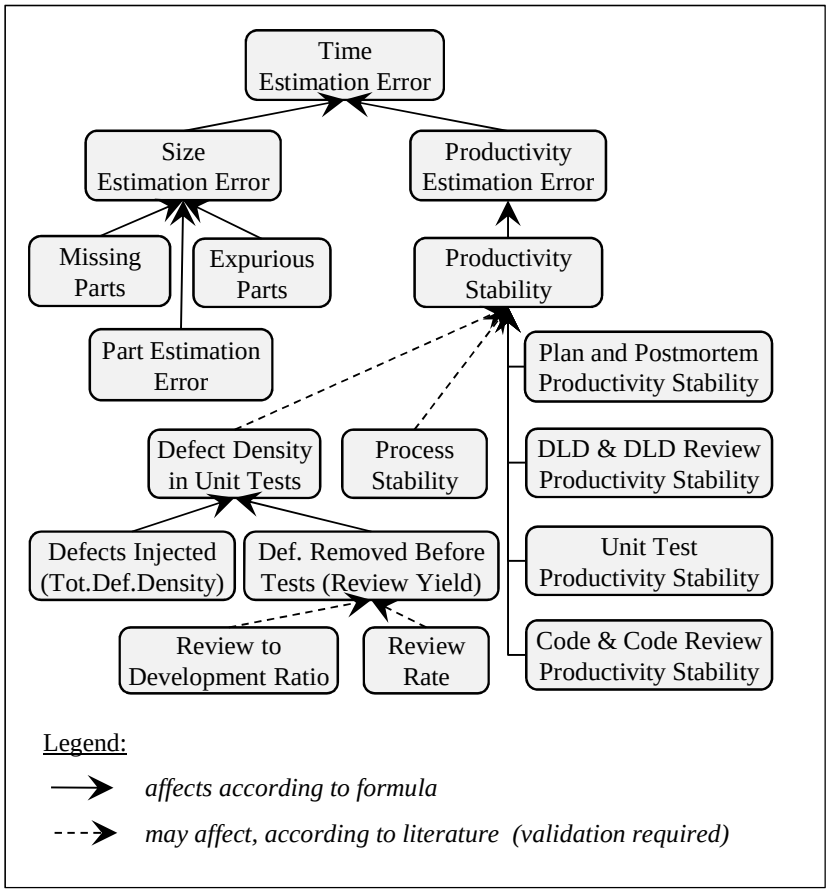


Figure 10: Performance Model for Identifying Causes of Estimation Problems

In order to identify the factors that, in turn, influence the SizeEE, one has to understand further the PROBE method. In this method, in order to arrive at a size estimate of a deliverable, one has to identify a list of parts that are needed to build the deliverable (which may be depicted in a so-called conceptual design); define the type of each part (I/O, calculation, etc.); estimate the relative size of each part in a T-shirt size (small, medium, large, etc.); use a so-called relative size table, based on historical data, to convert relative sizes to numerical sizes; and sum up the numerical sizes to obtain a size estimate of the deliverable. Hence, possible causes for a poor SizeEE (as summarized in Figure 10 and formalized by Duarte [Duarte 2012]) are missing parts (i.e., parts that were not planned but turned out to be needed); expurious (that is, extraneous or spurious) parts (i.e., parts that were planned but turned out to be unnecessary); and errors in the size estimates of the parts whose need was correctly identified.

In the PROBE method, productivity estimates are based on historical productivity, so the quality of the former depends on the stability of the productivity, as indicated in Figure 10. We calculate the productivity stability up to the project under analysis, based on the actual productivity of that project and past projects of the same developer (see Duarte's work for details [Duarte 2012]). In the PSP, time is recorded per process phase (Plan, Detailed Design, Detailed Design Review, Code, Code Review, Compile, Unit Test, and Postmortem), so we can compute the productivity stability per phase (see Figure 10). This provides useful information to locate instability problems in specific phases. It should be noted that the scope of the PSP is the development of individual

programs or components, which is the reason why Requirements, High Level Design, and System Testing phases are not included—they can be found in the more complete TSP. We also ignore the Compile phase, because we intend to analyze projects developed with programming languages without a separate Compile phase (like Java and C#).

On the other hand, process changes can cause productivity disturbances: usually, after a process change, productivity decreases and later on returns to the original productivity or a new productivity. This is an important factor when analyzing PSP training data, because the process is changed at least twice along the training (from PSP0 to PSP1 and PSP2), and existing data from PSP courses show that productivity variations follow. That's why we indicated Process Stability as potentially affecting Productivity Stability in Figure 10.

PSP literature states that the effort for finding and fixing defects through reviews is much more predictable than through testing, because in the former case the review effort is proportional to the size of the product under review and defects are immediately located, while in the latter case the time needed to locate defects is highly variable [Humphrey 2005]. High defect density in tests is considered a common cause of predictability problems. That's why we indicated in Figure 10 the defect density in unit tests as a factor that may affect productivity stability.

In turn, the defect density in unit tests is affected by the overall number of defects injected (and found) and the percentage of defects removed in reviews prior to testing, (i.e., the review yield).

The review yield is a lagging indicator of the quality of reviews (i.e., detailed design reviews and code reviews). PSP literature also states that the time spent in reviewing a work product, measured in relation to its size or the time spent developing it, is a leading indicator of the review yield, based on existing PSP data [Humphrey 2005]. A recent study confirms that the review rate is a significant factor affecting review effectiveness (measured as the percentage of defects that did not escape the reviews) [Kemerer 2009]; the recommended review rate of 200 LOC/hour or less was also found to be an effective rate for individual reviews, identifying nearly two-thirds of the defects in design reviews and more than half of the defects in code reviews. Another study in an industrial setting shows an improvement in inspection effectiveness when the review rate is reduced to a value closer to the recommended value [Ferreira 2010]. Another way of assessing the time spent in reviewing a work product is by comparing the review time with the production time. Humphrey's Software Quality Profile, based on extensive data from the PSP, presents a profile of software processes that generally produce high-quality products; among other aspects, design review time should be greater than 50% of design time, and code review time should be greater than 50% of coding time [Humphrey 2009]. For those reasons, we indicate in Figure 10 that the review yield may be affected by the review to development ratio and the review rate (size reviewed/time).

All the indicators are summarized in Table 7.

Table 7: Performance Indicators

Indicator (Abbreviation)	Formula
Time Estimation Error (TimeEE)	$\frac{Actual\ Time - Estimated\ Time}{Estimated\ Time}$
Size Estimation Error (SizeEE)	$\frac{Actual\ Size - Estimated\ Size}{Estimated\ Size}$

Indicator (Abbreviation)	Formula
Part Estimation Error (PartEE)	$\frac{\sum(Identified\ Part\ Actual\ Size)}{\sum(Identified\ Part\ Estimated\ Size)}$
Missing Parts (MP)	$\frac{\sum(Missing\ Part\ Actual\ Size)}{\sum(Missing\ or\ Identified\ Part\ Actual\ Size)}$
Expurious Parts (EP)	$\frac{\sum(Expurious\ Part\ Estimated\ Size)}{\sum(Expurious\ or\ Identified\ Part\ Estimated\ Size)}$
Productivity Estimated Error (PEE)	$\frac{Actual\ Productivity - Estimated\ Productivity}{Estimated\ Productivity}$
Productivity Stability (ProdS)	$\frac{Current\ Productivity - Historical\ Productivity}{Historical\ Productivity}$
Process Stability (ProcS)	True, if the process used in the current project is the same as in the previous project; false, otherwise.
Defect Density in Unit Test (DDUT)	$\frac{\#Defects\ found\ in\ Unit\ Test}{Actual\ Size\ (KLOC)}$
Total Defect Density (TotalDD)	$\frac{\#Defects\ found\ in\ all\ phases}{Actual\ Size\ (KLOC)}$
Review Yield (RY)	$\frac{Defects\ removed\ in\ reviews}{\#Defects\ found}$
Review Rate (RR)	$\frac{Actual\ Size}{Review\ Time}$
Review to Development Ratio (R2D)	$\frac{Actual\ Time\ of\ Reviews\ (DLDR + CR)}{Actual\ Time\ of\ Development\ (DLD + CODE)}$

3.2.2 Control Limits

In order to recognize performance problems, for each performance indicator shown in Figure 10 we defined control limits for classifying values of the performance indicator into three categories: green—no performance problem; yellow—a possible performance problem; red—a clear performance problem. All the defined control limits can be found in Table 8. These control limits are based on recommended values usually considered for PSP projects, present in the literature [Humphrey 2005] or orally stated by PSP instructors. Nevertheless, these control limits can be easily configured by the users of our PSP PAIR tool, described in Section 3.3.

Table 8: Control Limits for the Performance Indicators

Indicator	Green	Yellow	Red
TimeEE	|TimeEE|≤ 20%	20% < |TimeEE| ≤ 30%	|TimeEE| > 30%
SizeEE	|SizeEE| ≤ 20%	20% < |SizeEE| ≤ 30%	|SizeEE| > 30%
PartEE	|PartEE| ≤ 10%	10% < |PartEE| ≤ 20%	|PartEE| > 20%
MP	0 ≤ MP ≤ 10%	10% < MP ≤ 20%	MP > 20%
EP	-10% ≤ EP ≤ 0	-20% ≤ EP < -10%	EP < -20%
PEE	|PEE| ≤ 20%	20% < |PEE| ≤ 30%	|PEE| > 30%
ProdS	|ProdS| ≤ 20%	20% < |ProdS| ≤ 30%	|ProdS| > 30%
ProcS	ProcS = True		ProcS = False
DDUT	0 ≤ DDUT ≤ 5	5 < DDUT ≤ 10	DDUT > 10

Indicator	Green	Yellow	Red
TotalDD	0≤TotalDD≤50	50 < TotalDD ≤ 100	TotalDD > 100
RY	70%≤RY≤100%	60% ≤ RY < 70%	RY < 60%.
RR	100 ≤ RR ≤ 200	50 ≤ RR < 100 or 200 < RR ≤ 250	0 ≤ RR < 50 or RR > 250
R2D	R2D ≥ 0.5	0.25 ≤ R2D< 0.5	0 ≤ R2D< 0.25

3.2.3 Work in Progress: Improvement Actions

To be able to readily recommend improvement actions for the identified performance problems, we need to build a knowledge base of possible remedial actions. We put together an initial set of candidate recommendations (see Table 9), but our goal is to obtain suggestions and feedback from the PSP/TSP community to arrive at a set of consensual recommendations that can be used in practice.

Table 9: Preliminary List of Problems and Improvement Actions

Indicator	Problems identified and Suggested Improvement Actions
Missing Parts	Problem with the identification of the parts. Recommended actions • A more careful Conceptual Design is recommended, with more detail or time spent in design.
Expurious Parts	Problem with the identification of the parts. Recommended actions • A more careful Design is recommended, with more detail or time spent in design.
Part Estimation Error	Problem in the relative-size table and/or problem with the identification of the parts. Recommended actions • Review the types of the parts, their relative size, and related topics.
Productivity Stability – Plan and Postmortem	Problem with the changes in the way of working. Recommended actions • Try to keep a stable productivity and check what has been modified lately. • If the productivity has increased, try to keep it stable at that new value in future projects.
Productivity Stability – DLD and DLD Review	Problem with the changes in the way of working. Recommended actions • Try to keep a stable productivity and check what has been modified lately. • If the productivity has increased, try to keep it stable at that new value in future projects.
Productivity Stability – Code and Code Review	Problem with the changes in the way of working. Recommended actions • Try to keep a stable productivity and check what has been modified lately. • If the productivity has increased, try to keep it stable at that new value in future projects.
Productivity Stability – Unit Test	Problem with changes in the way of working. Recommended actions • Try to keep a stable productivity and check what has been modified lately. • If the productivity has increased, try to keep it stable at that new value in future projects.
Process Stability	Recommended actions • Try to keep the process stable. • Changing the process usually leads to a variation in productivity.
Total Defect Density	Recommended actions • Improve the process quality. • Do an analysis on the more frequent and more expensive errors in order to not repeat them; define preventive actions.
Review Rate	Recommended actions • If the value is too high, it is necessary to reduce it by doing the review more carefully and slowly. • If it is too slow (low value), do the review in a more focused and efficient way. • Check if the artifacts are understandable.

Indicator	Problems identified and Suggested Improvement Actions
Review to Development Ratio	Recommended actions • If the value is too high, it is necessary to do the review more carefully and slowly or code faster and more efficiently. • If the value is too low, do the review in a more focused and efficient way or take your time coding. • Check if the artifacts are understandable.

3.3 The PSP PAIR Tool

The tool developed was named PSP PAIR after PSP Performance Analysis and Improvement Recommendation.[1] It analyzes the performance data produced by individual PSP developers along several projects, as recorded in the PSP Student Workbook supplied by the Software Engineering Institute. The analysis is based on a performance model such as the one described in the previous section, represented in an XML file for easier configuration and extensibility. This way, other PIs and performance models can be used in the future without the need to change the PSP PAIR tool.

The home page of the PSP PAIR tool is shown in Figure 11. It has options to load a new database file (a Microsoft Access file exported by the PSP Student Workbook) and load a new performance model in XML. Once a database file is loaded, the tool presents a list of projects contained in it, so that the user can select the projects to be analyzed. The results of the data analysis are presented in a new window, as shown in Figure 12.

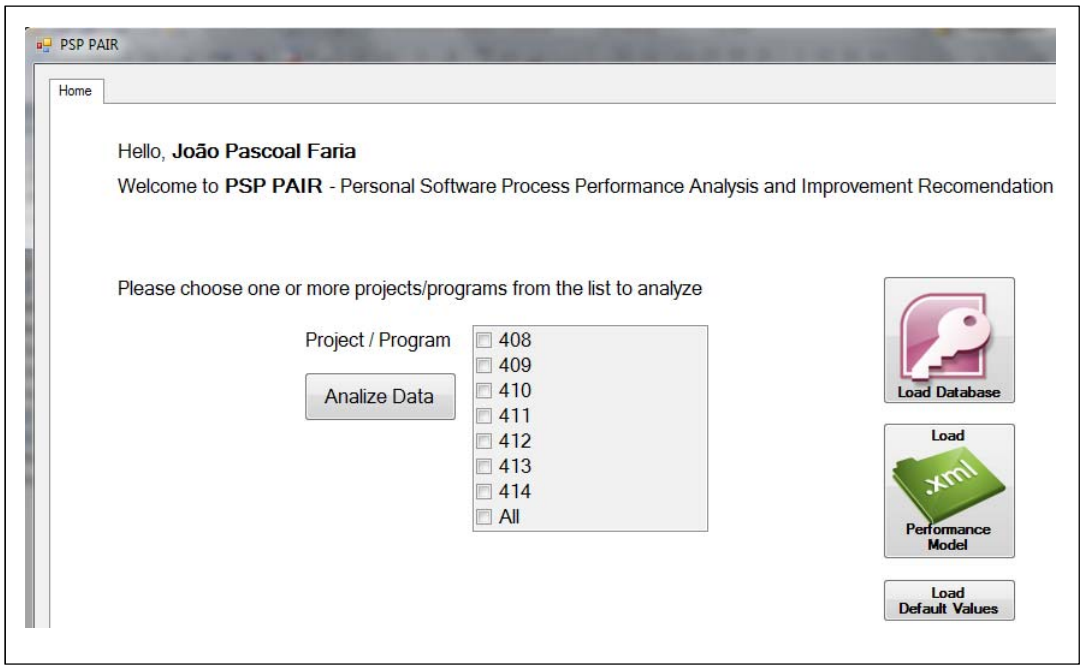
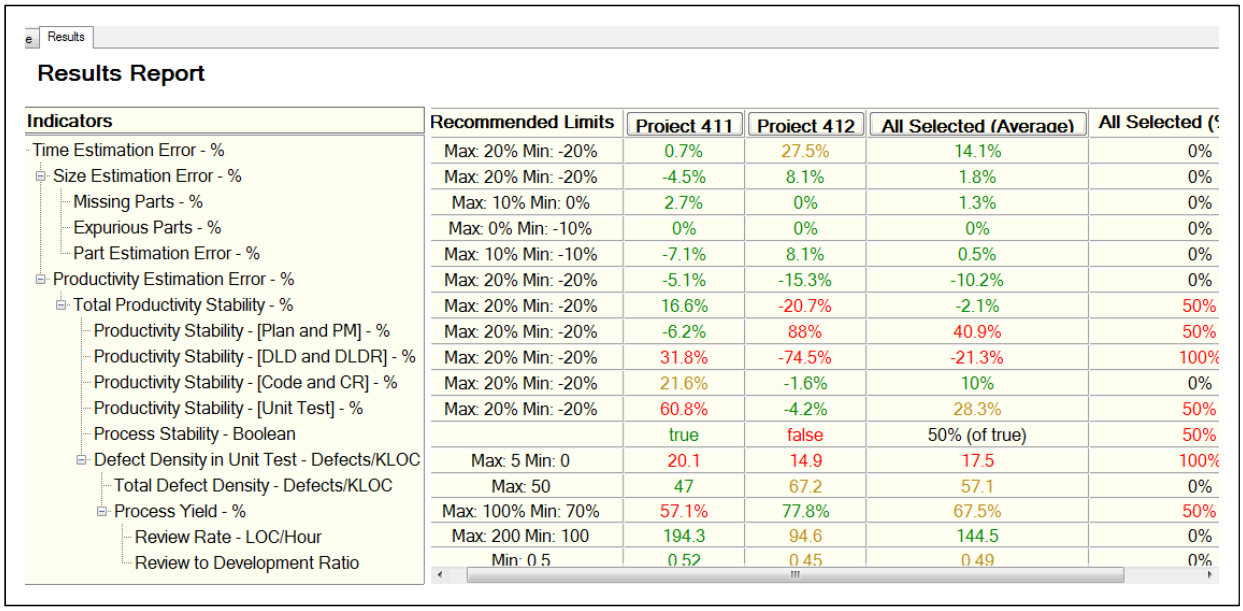


Figure 11: PSP PAIR Home

[1] The PSP PAIR tool can be downloaded through the Faculdade de Engenharia da Universidade do Porto website (http://paginas.fe.up.pt/~ei06089/wiki/lib/exe/fetch.php?media=projects:psp_pair_-_installer.zip).



Recommended Limits	Project 411	Project 412	All Selected (Average)	All Selected (%
Max: 20% Min: -20%	0.7%	27.5%	14.1%	0%
Max: 20% Min: -20%	-4.5%	8.1%	1.8%	0%
Max: 10% Min: 0%	2.7%	0%	1.3%	0%
Max: 0% Min: -10%	0%	0%	0%	0%
Max: 10% Min: -10%	-7.1%	8.1%	0.5%	0%
Max: 20% Min: -20%	-5.1%	-15.3%	-10.2%	0%
Max: 20% Min: -20%	16.6%	-20.7%	-2.1%	50%
Max: 20% Min: -20%	-6.2%	88%	40.9%	50%
Max: 20% Min: -20%	31.8%	-74.5%	-21.3%	100%
Max: 20% Min: -20%	21.6%	-1.6%	10%	0%
Max: 20% Min: -20%	60.8%	-4.2%	28.3%	50%
	true	false	50% (of true)	50%
Max: 5 Min: 0	20.1	14.9	17.5	100%
Max: 50	47	67.2	57.1	0%
Max: 100% Min: 70%	57.1%	77.8%	67.5%	50%
Max: 200 Min: 100	194.3	94.6	144.5	0%
Min: 0.5	0.52	0.45	0.49	0%

Figure 12: PSP PAIR Performance Results Report

The PIs are shown in a tree view, according to the relationships defined in the performance model. The recommended control limits for each indicator are always shown on the first column, followed by the values of the PIs in the selected projects, colored according to the control limits. The last two columns display the average value and the percentage of red colored values for the selected projects.

In the example shown in Figure 12 (based on real PSP data), we can quickly spot some performance problems (colored red). In project 411, there are problems in the following indicators: productivity stability in DLD and DLDR; productivity stability in unit tests (PSUT); DDUTs—a possible cause for the previous productivity stability problems according to our performance model; and process yield (same as review yield)—a possible cause for the previous DDUT problems according to our performance model. In project 412, there are problems in the following indicators: productivity stability, namely in the Plan, PM, DLD and DLDR phases; process stability—a possible cause for the previous productivity stability problems according to our performance model; and defect density in unit test—another possible cause for the previous productivity stability problems according to our performance model.

Recommended improvement actions to address the performance problems identified in a single project or all the selected projects are presented by the tool in a new window, using the same color scheme as for the performance values (yellow or red, according to the color of the triggering performance value), as shown in Figure 13 (please note that the recommended improvement actions at this time are very preliminary). This aids understanding of which problems should be addressed first.

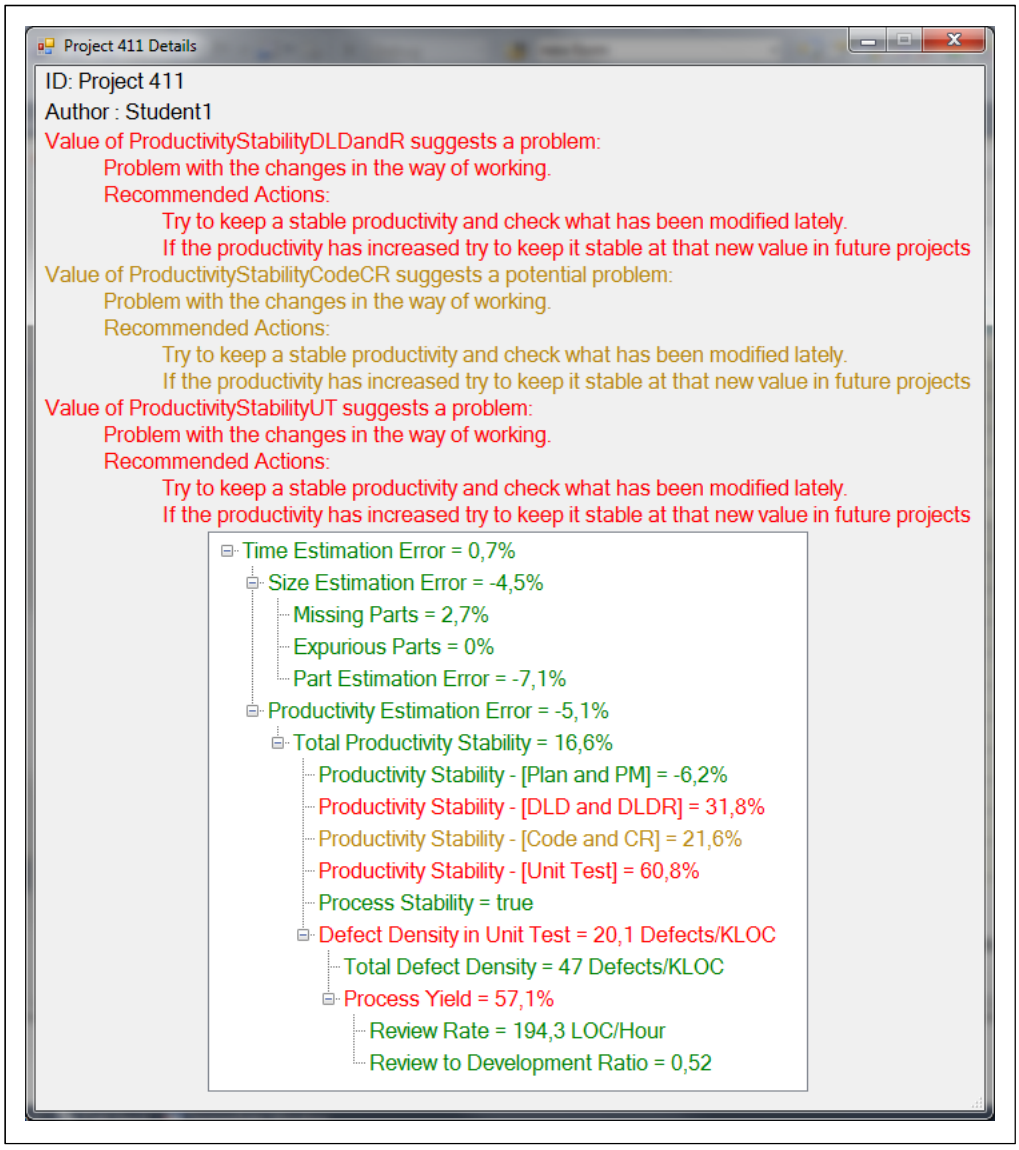


Figure 13: PSP PAIR Recommended Improvement Actions for a Sample Project

3.4 Evaluation

In the end of the PSP Advanced course, developers have to analyze their personal performance along a series of seven projects and document their findings and suggestions for improvement in a Performance Analysis Report (PAR). The main result of this analysis is a prioritized and quantified set of Process Improvement Proposals (PIPs). The goal of our research work is to help automate this kind of analysis. So, to partially evaluate our approach, we compared the results produced by our PSP PAIR tool with the results of a manual analysis conducted and documented by an advanced PSP user, for the same project data.

A synthesis of the results obtained with both approaches is presented in Table 10 and Table 11. The results are quite similar: the conclusions arrived at in the PAR document (such as problems identified) were consistently present in the problems identified by the PSP PAIR. It should be noted that some normalization had to be made regarding the presentation of the results

documented in the PAR. Further details are provided in the publication titled "Automated Software Processes Performance Analysis and Improvement Recommendation" [Duarte 2012].

One of the major advantages of the automated analysis in this case is that it was almost instantaneous, while the manual performance analysis took more than eight hours (PSP developers have to record the time spent in producing the PAR).

But the manual analysis also has some advantages in this case: by taking into account additional quantitative and qualitative information, it was possible to analyze in greater detail the cause for productivity instability in DLD and DLDR and propose corresponding improvement actions. This does not diminish the value of our tool, because it correctly points out problems in DLD and DLDR productivity stability. Based on that information, a more focused manual analysis could be subsequently performed.

Table 10: Results Assessment: Problems

Manual Analysis (Performance Analysis Report)	**PSP PAIR**
Problems	Problems
• Time estimation accuracy (same as TEE) • Productivity, namely at DLD phase • Defect density in UT	• Value of Productivity Stability in DLD + DLDR suggests a problem. o Problem with the changes in the way of working • Value of Total Defect Density suggests a potential problem. Based on the average of all the projects, there were problems identified at o Time estimation error o Total productivity stability o Productivity stability at DLD and DLD review o Defect density in UT

Table 11: Results Assessment: Improvement Actions

Manual Analysis (Performance Analysis Report)	**PSP PAIR**
Process Improvement Proposals	Recommended Actions
• Stabilize and make more efficient the DLD process to improve and stabilize productivity: o Avoid expensive DLD representations (e.g., equations in Word) o Avoid long DLD documents (compared to code size) o Do Logic Specifications that can be used as code comments and can be easily written • Widen the size range to have more basis for estimations.	• Try to keep a stable productivity and check what has been modified lately. o If the productivity has increased try to keep it stable at that new value in future projects. • Improve the process quality. o Do an analysis on the more frequent and more expensive errors in order to not repeat them; define preventive actions.

3.5 Presentation of Evidence—Impact of Process Changes on Productivity Stability

In this section, we provide empirical evidence in favor of the cause-effect relationship between process stability and productivity stability (see Figure 10), based on an experiment conducted by one of the authors in an academic setting.

The developers were nearly 100 students from the 2009/10 edition of the "Software Engineering"[2] course of the Integrated Master Program in Informatics Engineering at University of Porto. Along with other tasks, students had to develop a sequence of four programs and collect a set of performance measures, following increasingly elaborated development processes (see Table 11). The process scripts and performance measures were based on PSP with some changes due to resource limitations: students could work alone or in pairs (pair programming); only the simplest PROBE estimation method variant was used (averaging); in case of problems, students were asked to resubmit their work only in the first assignment; students got a mark from each assignment; there was no Compile phase.

Table 12: Overall Statistics of the Experiment

Program	Process Changes	Number of Submissions	Number of Developers	Mean Size (LOC)	Mean Time (min.)	Mean # Defects
Program 1 - mean and standard deviation calculations; linked list.	Baseline process (with time/size/defects performance measures)	55	97	160	160	3.0
Program 2 - linear regression calculations (reusing P1)	PLAN: Size and effort Estimation (with PROBE method)	55	96	136	150	2.0
Program 3 - Student's t-ditribution calculations	CODE: Coding standards (comments, etc.) CR: new phase UT: automated with xUnit	53	93	109	205	2.8
Program 4 - linear regression with significance and prediction interval (reusing P1, P2, P3)	DLD: UML diagrams (class, sequence) DLDR: added this phase PM: Proc. Improv. Proposals	46	79	71	184	1.7

Based on the summary data of each submission, namely the time spent per process phase (computed from the time recording log) and the final program size (new and changed lines of code, excluding test code, comments and blank lines), we produced the chart in Figure 14.

Not surprisingly, the greatest changes in average productivity per phase are related to process changes. When additional steps are introduced in a process phase, there is a significant variation (degradation in this case) of the average productivity of that phase. By contrast, when the

[2] https://sigarra.up.pt/feup_uk/DISCIPLINAS_GERAL.FORMVIEW?P_ANO_LECTIVO=2009/2010&P_CAD_CODIGO=EIC0024&P_PERIODO=1S

definition of the process phase is unchanged, the productivity remains mostly stable (in most of the cases, with a slight improvement). The results of the pairwise ANOVA analysis, presented in Table 13, confirm the statistical significance of the differences in productivity per phase.

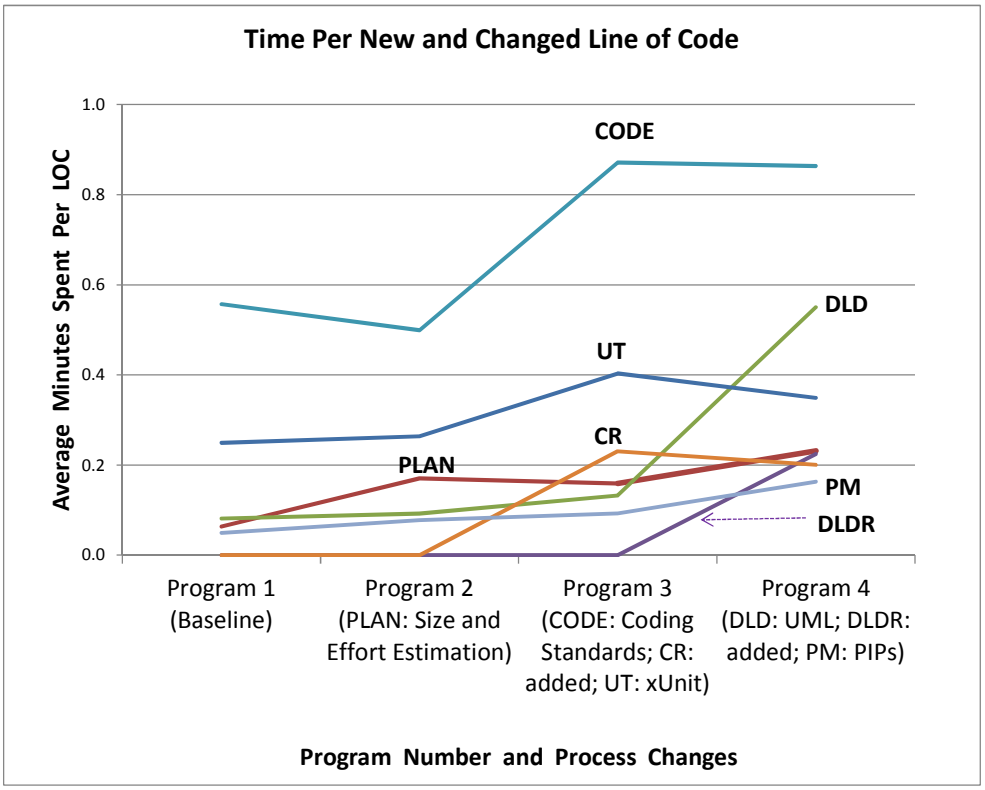


Figure 14: Evolution of the Mean Productivity Per Phase (in Minutes Spent Per New or Changed LOC) Along the Four Program Assignments

Table 13: Results of Pairwise ANOVA Analysis, Highlighting Statistically Significant Differences in Mean Productivity Per Phase Between Pairs of Program Assignments, with a 95% Confidence Level (p value < 0.05)

p value	PLAN	DLD	DLDR	CODE	CR	UT	PM
P1 - P2	**0.0027**	0.9998	-	0.8490	-	0.9882	0.3837
P1 - P3	**0.0313**	0.8277	-	**0.0102**	-	0.1054	0.2107
P1 - P4	**0.0000**	**0.0000**	-	0.0618	-	0.1838	**0.0000**
P2 - P3	0.9127	0.8727	-	**0.0009**	-	0.0554	0.9780
P2 - P4	**0.0063**	**0.0000**	-	**0.0081**	-	0.1042	**0.0000**
P3 - P4	**0.0010**	**0.0000**	-	0.9457	0.9313	0.9968	**0.0000**

A more detailed, phase-by-phase analysis follows.

PLAN—While in program 1 students estimated size and time empirically, in program 2 and subsequent programs they were asked to apply the PROBE method (described in Section 3). Not

surprisingly, the time spent per LOC increased significantly in program 2 and remained stable thereafter, with a small degradation in program 4 (possibly because of the high amount of reuse).

DLD—In all programs, students were asked to spend some time in design, but only in program 4 they were asked to draw computer-based class diagrams and sequence diagrams. Not surprisingly, this caused a significant leap in design time, which was not recovered in the Code phase. (Investment in explicit UML designs may only compensate for complex programs, but our goal was to let students learn the techniques in the small.)

CODE—Coding standards, requiring code comments in class and method headers, were introduced in program 3. Not surprisingly, the time spent per LOC (not counting comments) increased in program 3. Otherwise, there is a tendency for improvement.

CR—Code reviews were introduced only in program 3. From program 3 to 4 there is a small productivity improvement in doing code reviews. We didn't expect that the time invested in code reviews would be recovered significantly in the unit test effort, because we asked students to focus code reviews on aspects complementary to tests—mainly adherence to coding standards and exception handling.

UT—Students were asked to unit test all programs (at least for the test cases given), but only in program 3 and subsequent programs were they asked to automate their tests with JUnit. Not surprisingly, this required additional effort from the developers (test automation effort is usually recovered only after a number of repetitions of test execution, and we did not count test code in size measurement). From programs 3 to 4 there is a small productivity improvement in the unit test phase.

PM—In all programs, in the postmortem phase developers had to measure the final program size (using a tool) and perform final data consistency and completeness checks. Additionally, in program 4 they were asked to produce at least one Process Improvement Proposal. Not surprisingly, this caused the highest variation in productivity of this phase.

3.6 Presentation of Evidence—Impact of Defect Density in Unit Tests on Productivity Stability

We next provide evidence partially supporting the cause-effect relationship between the Defect Density in Unit Tests and Productivity Stability (see Figure 10), based on results from the same experiment presented previously.

In all assignments, students were required to record in a defect log the defects they found and fixed, including, for each defect, the phase in which it was injected (e.g., CODE), the phase in which it was removed (found and fixed) (e.g., UT), and the fix time (rework time).

For each submission, we computed the Defect Density in Unit Test (DDUT) and the Productivity Stability (ProdS) (see definitions in Table 7). Since ProdS is not defined for the first project of each team, 67 submissions from the 209 submissions were excluded from consideration (all the submissions of program 1 plus additional submissions after changes in team composition). We also excluded from consideration 18 more submissions with obvious data quality problems in defect recording (in most of the cases, because of a long test time and no defects recorded). In the

end, we got 125 data points with the distribution shown in Figure 15. The number of data points (submissions) in is shown below each class.

Unfortunately, the abnormal distribution of the data points with zero DDUT, as compared to the tendency shown by the remaining data points, strongly suggests that data quality problems still reside in these data points (such as students that did not record their defects). The data points with non-zero DDUT show a tendency for a productivity increase (when compared with historical productivity) when DDUT is small and a tendency for decrease when DDUT is high.

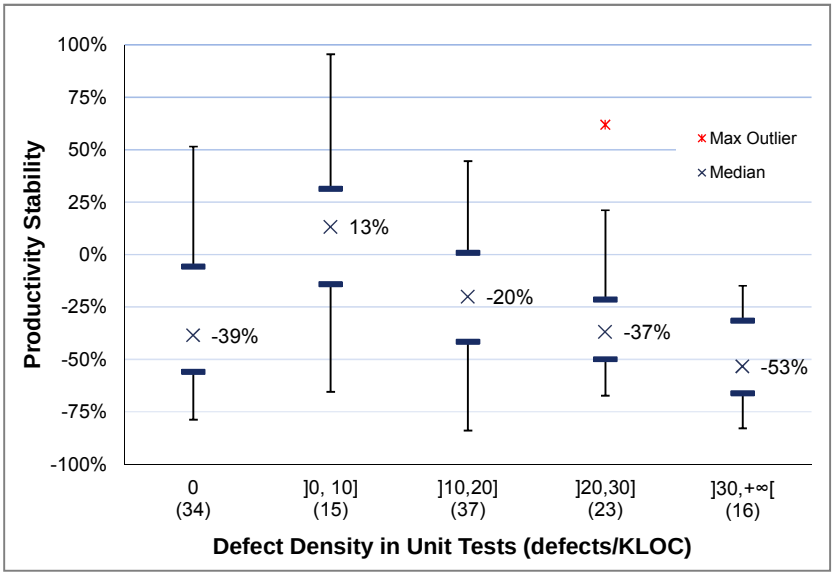


Figure 15: Box and Whisker Plot Showing the Relationship Between the Defect Density in Unit Tests and the Productivity Stability

This tendency is not surprising. If, for each team, the number of defects has significant variations along projects and the defect fix time (rework) has a non-negligible value with significant variations, then productivity fluctuations will follow with a tendency as exhibited by the data set analyzed. To confirm this explanation, we computed the statistical behavior of the defect fix time. According to Figure 16, the majority of defects are removed (found and fixed) in unit test (UT), where the median and dispersion value of the fix time (rework time) are much higher than in code reviews (CRs). The number of data points (defects) is shown under each removal phase.

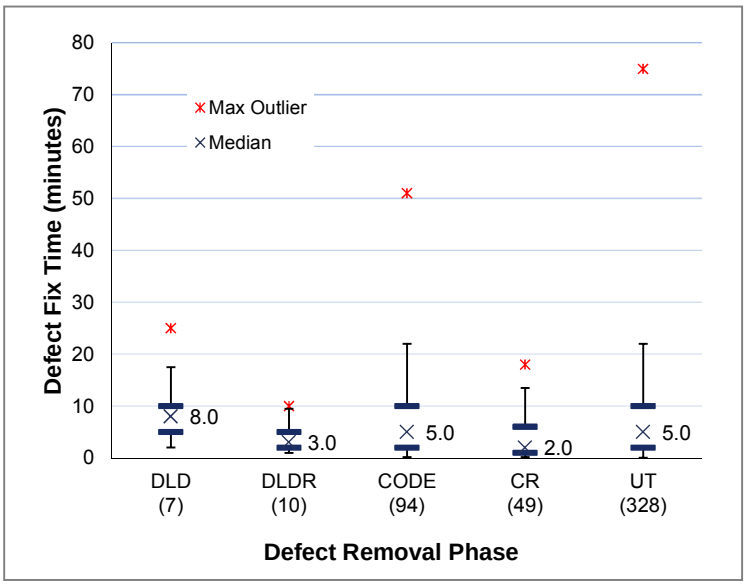


Figure 16: Box and Whisker Plot Showing the Relationship Between the Defect Removal Phase and the Defect Fix Time

3.7 Conclusions and Future Work

The experiments conducted so far show that the performance model and tool that we developed effectively aid in accelerating the identification of performance problems and determination of improvement actions in the context of the PSP.

Although the work presented is limited to the analysis of time estimation performance of PSP developers, other performance aspects are also analyzed because of the impact of the time estimation performance, and the tool was designed so that it can be easily configured and extended for analyzing other performance aspects and development processes.

As future work we intend to

- validate further the dependencies between performance indicators that do not result directly from the formulas, based on other data sources (namely, based on data collected from hundreds of developers that attended PSP training courses)
- rank the recommended improvement actions presented to the user, according to a prediction of their cost and benefit (partially computed from the same performance model)
- take advantage of the feedback from the community of PSP users (and PSP PAIR users) in the ranking of improvement actions and evolution of the catalog of improvement actions
- accept other input formats, other than the format exported by the PSP Student Workbook
- extend the performance model for other performance indicators of interest in the scope of the PSP
- extend the approach for analyzing data produced in the context of other development processes, namely in the context of the TSP
- integrate the tool and features presented in software as a service platform, under development by a national consortium, integrating project, process, and performance management

3.8 Acknowledgments

This work is funded in part by FEDER (*Fundo Europeu de Desenvolvimento Regional*) through the Portuguese ON.2 Program (*Programa Operacional Região do Norte*), under project reference SI IDT - 21562/2011.

3.9 Author Biographies

César Duarte
Consultant in Software Engineering
Strongstep – Innovation in Software Quality

César Duarte is a software engineer and holds a Master's Degree in Informatics and Computing Engineering. He graduated from the Faculty of Engineering, University of Porto (FEUP) in Portugal and is currently working as a consultant in software engineering at Strongstep – Innovation in Software Quality. He also worked as a research assistant at ESB - Universidade Católica Portuguesa - Porto. His specialties and research interests are related to software engineering, software process improvement, PSP, and TSP.

João Pascoal Faria
Assistant Professor and Researcher
INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto

João Pascoal Faria received his PhD in Electrical and Computer Engineering from the Faculty of Engineering of the University of Porto in 1999 and is currently an assistant professor at the university in the Department of Informatics Engineering and a researcher at INESC Porto. He is vice-president of the Sectorial Commission for the Quality in Information and Communication Technologies (CS03) from the Portuguese Quality Institute (IPQ). In the past, he worked with several software companies and was a co-founder of two others. He has more than 20 years of experience in education, research, and consultancy in several software engineering areas. He is the main author of a rapid application development tool (SAGA), based on domain specific languages, with more than 20 years of market presence and evolution (1989-2011). Since 2008, he is a certified PSP Developer, authorized PSP Instructor, and trained TSP Coach by the Software Engineering Institute of Carnegie Mellon University. He is currently involved in research projects, supervisions, and consulting activities in the areas of model-based testing, model-driven development, and software process improvement.

Mushtaq Raza
PhD student
MAP-i Doctoral Program, University of Porto, Portugal

Mushtaq Raza is a PhD student in the MAP-i doctoral program at the University of Porto, Portugal. He is also serving Abdul Wali Khan University in Mardan, Pakistan as an assistant professor. His research interests include PSP, Global Software Development (GSD), and Usability Engineering.

Pedro Castro Henriques
Director and Senior Consultant
Strongstep – Innovation in Software Quality

Pedro Castro Henriques has a 5-year degree in software engineering and a post-graduate degree in technology entrepreneurship and commercialization. His thesis was on information systems strategic planning for the health sector. He began his career 12 years ago as a software engineer at Q-Labs Lund/DNV and soon became a consultant working in nine European countries. After his

international experience he returned to Portugal and founded the Oporto software engineering alumni association (which now has more than 1600 members). Afterward he further specialized in process improvement, implementation, and certification in software quality. His studies in internationalization and innovation of companies and his participation in an entrepreneurship semester in Porto Business School grounded his career in this critical area. Extremely committed to innovation and entrepreneurship, he co-founded Strongstep – Innovation in Software Quality and Portic in 2010. He is currently the director of Strongstep and president of Portic. He was a facilitator in bringing SEPG Europe to Portugal for the first time in 2010. He is also a professor at FEUP in the Services Management Engineering Master, focusing on service requirements, and an invited lecturer in the Software Quality and Tests at Master.

3.10 References/Bibliography

[Burton 2006]
Burton, D. & Humphrey, W. S. "Mining PSP Data." *Proceedings of the TSP Symposium*. New Orleans, LA, September 2006. http://www.sei.cmu.edu/tspsymposium/2009/2006/mining.pdf

[C. S. D. Laboratory 2010]
C. S. D. Laboratory, 2010. *Hackystat.* http://code.google.com/p/hackystat

[Davis 2003]
Davis, B. N. & Mullaney, J. L. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014). Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Design Studio Team 1997]
Design Studio Team, East State Tennessee University, 1997. PSP Studio. http://csciwww.etsu.edu/psp/

[Duarte 2012]
Duarte, Cesar Barbose. "Automated Software Processes Performance Analysis and Improvement Recommendation." MSc Thesis, Informatics and Computing Engineering, University of Porto, 2012.

[Ferreira 2010]
Ferreira, Andre L.; Machado, Ricardo J.; Costa, Lino; Silva, José G.; Batista, Rui F.; & Paulk, Mark C. "An Approach to Improving Software Inspections Performance," 1-8. *ICSM'10 Proceedings of the 2010 IEEE International Conference on Software Maintenance.* Timisoara, Romania, September 2010. IEEE, 2010. ISBN 978-1-4244-8640-4.

[Humphrey 2009]
Humphrey, Watts S. *The Software Quality Profile*. Software Engineering Institute, Carnegie Mellon University, 2009. http://www.sei.cmu.edu/library/abstracts/whitepapers/qualityprofile.cfm

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Humphrey 2002]
Humphrey, W. S. "Personal Software Process (PSP)." *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., 2002.

[Kemerer 2009]
Kemerer, Chris & Paulk, Mark. "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data." *IEEE Transactions on Software Engineering 35*, 4 (July–August 2009): 534-550.

[Nasir 2005]
Nasir, M. H. N. M. & Yusof, A. M. "Automating a modified personal software process." *Malaysian Journal of Computer Science 18*, 2 (December 2005): 11–27.

[Pomeroy-Huff 2012]
Pomeroy-Huff, M.; Cannon, R.; Chick, T. A.; Mullaney, J.; & Nichols, W. *The Personal Software Process(PSP) Body of Knowledge, Version 2.0* (CMU/SEI-2009-SR-018). Software Engineering Institute, Carnegie Mellon University, 2009.
http://www.sei.cmu.edu/library/abstracts/reports/09sr018.cfm

[Shen 2011]
Shen, Wen-Hsiang; Hsueh, Nien-Lin; Lee, & Wei-Mann. "Assessing PSP effect in training disciplined software development: A Plan–Track–Review model." *Information and Software Technology 53* (February 2011): 137–148.

[Shin 2007]
H. Shin, H.; Choi, H.; & Baik, J. "Jasmine: A PSP Supporting Tool." *Software Process Dynamics and Agility 4470*, pp. 73–83. Edited by Q. Wang, D. Pfahl, & D. Raffo. Springer-Verlag, 2007.

[Sison 2005]
Sison, R. "Personal Software Process (PSP) assistant," 687–696. *Proceedings of the 12th Asia-Pacific Software Engineering Conference* (*APSEC '05*). Washington, D.C., February 2005. IEEE Computer Society, 2005.

[Software Process Dashboard Initiative, 2011]
The Software Process Dashboard Initiative. "Functionality for Individuals," 2011.
http://www.processdash.com/functionality-personal

4 PSP_{DC}: An Adaptation of the PSP to Incorporate Verified Design by Contract

Silvana Moreno, Universidad de la República
Álvaro Tasistro, Universidad ORT Uruguay
Diego Vallespir, Universidad de la República

4.1 Introduction

Personal Software Process (PSP) incorporates process discipline and quantitative management into the software engineer's individual development work. It promotes the exercise of careful procedures during all stages of development with the aim of achieving high quality of the final product and thereby also increasing the individual's actual productivity [Humphrey 2005, 2006].

Formal methods, in turn, use the same methodological strategy, namely emphasizing care in the procedures of development (as opposed to relying on testing and debugging.) In fact they establish a radical requirement in this respect, which consists of mathematically proving that the programs satisfy their functional requirements.

In this paper we investigate how to integrate the use of formal methods into PSP, by formulating a new version of PSP.

Design by Contract (DbC) is a technique devised (and patented) by Bertrand Meyer for designing components of a software system by establishing their conditions of use and behavioral requirements in a formal language [Meyer 1992]. This language is seamlessly integrated with the programming language so that the specified conditions can actually be evaluated at run-time, which allows, among other things, the ability to handle violations by means of a system of exceptions. When appropriate techniques and tools are incorporated that allow proving that the components satisfy the established requirements, it indicates the use of a formal method usually called Verified Design by Contract (VDbC.) This is the method we propose to consider for integration with PSP.

Our alternative version of PSP is called PSP_{DC}. With respect to ordinary PSP, it incorporates new phases and modifies others as well as adding to the infrastructure new scripts and checklists. Specifically, we add the phases of Formal Specification, Formal Specification Review, and Formal Specification Compile. In addition we modify the Compile phase into a new phase called Compile and Proof. The general idea is to supplement the design with formal specifications of the components (which are produced in the first three new phases listed above) and the code with a formal proof that it matches the formal specifications (which are produced in the Compile and Proof phase.) This proof is to be carried out with the help of a tool akin to a verifying compiler that statically checks the logical correctness of the code in addition to the syntax.

We know of only two works in the literature that propose to combine PSP and formal methods. Babar and Potter combine Abrial's B Method with PSP into B-PSP [Babar 2005]. They add the phases of Specification, Auto Prover, Animation, and Proof. A new set of defect types is added and logs are modified to incorporate data extracted from the B machine's structure. The goal of this work is to provide the individual B developers with a paradigm of measurement and

evaluation that promotes reflection on the practice of the B method, inculcating the habit of recognizing causes of defects injected so as to be able to avoid these in the future.

Suzumori, Kaiya, and Kaijiri propose the combination of VDM and PSP [Suzumori 2003]. The Design phase is modified incorporating the formal specification in the VDM-SL language. Besides, the phases of VDM-SL Review, Syntax Check, Type Check, and Validation are added. One result arising from applications is that the use of VDM contributes to eliminate defect injection during design.

4.2 Formal Methods

Formal methods hold fast to the tenet that programs should be proven to satisfy their specifications. Proof is, of course, the mathematical activity of arriving at knowledge deductively (i.e., starting off from postulated, supposed, or self-evident principles and performing successive inferences, each of which extracts a conclusion out of previously arrived at premises).

In the application of this practice to programming we have among the first principles the so-called semantics of programs, which allow us to understand program code and thereby know what each part of the program actually computes. This makes it in principle possible to deductively ascertain that the computations carried out by the program satisfy certain properties. Among these properties are input-output relations or patterns of behavior that constitute a precise formulation of the so-called functional specification of the program or system at hand.

Formal logic, at least in its contemporary mathematical variety, has striven to formulate artificial languages into which it is possible to frame the mathematical activity. There should therefore, according to this aim, be a language for expressing every conceivable mathematical proposition and also a language for expressing proofs, so that a proposition is provable in this language if and only if it is actually true. This latter desirable property of the language is called its correctness. This kind of research began in 1879 with Frege [Frege 1967] with the purpose of making it undisputable whether a proposition was or was not correctly proven. Indeed, the whole point of devising artificial languages was to make it possible to automatically check whether a proposition or a proof was correctly written in the language. That is to say, the proofs accepted were to be so on purely syntactic (i.e. formal) grounds and, given the good property of correctness of the language, that was enough to ensure the truth of the asserted propositions.

Frege's own language turned out to be not correct and, for that reason mainly, shortly after its failure the whole enterprise of formal logic took a different direction, namely that of studying the artificial languages as mathematical objects to prove their correctness by elementary means. This new course was actually also destined to failure.

The overall outcome is nevertheless very convenient from an engineering viewpoint. We can go back to Frege's program, and nowadays we have technology that makes it feasible to develop formal proofs semi-automatically. The proof systems (or languages) are still reliable although not complete (i.e., not every true proposition will be provable). But again, there is no harm in the practice, and the systems are perfectly expressive from an engineering perspective. These advances allow us to define formal methods in software engineering as a discipline based on the use of formal languages and related tools for expressing specifications, and carrying out proofs of correctness of programs.

Notice that the semi-automatic process of program correctness proof is of course a static kind of checking. We can think of it as an extension of compilation, which not only checks syntax but also properties of functional behavior. Therefore it is convenient to employ the general idea of a semi-automatic verifying compiler to characterize the functionality of the tools employed within a formal methods framework.

DbC is a methodology for designing software proposed and registered by Bertrand Meyer [Meyer 1992]. It is based on the idea that specifications of software components arise, like business contracts, from agreements between a user and a supplier, which establish the terms of use and performance of the components. That is to say, specifications oblige (and enable) both the user and the supplier to certain conditions of use and a corresponding behavior of the component in question.

In particular, DbC has been proposed in the framework of Object Oriented Design (and specifically in the language Eiffel) and therefore the software components to be considered are classes. The corresponding specifications are pre- and post-conditions to methods, establishing respectively their terms of use and corresponding outcomes, as well as invariants of the class (i.e., conditions to be verified by every visible state of an instance of the class). In the original DbC proposal, all the specifications are written in the language Eiffel itself and are computable (i.e., they are checkable at run-time).

Therefore, DbC within Eiffel provides at least the following:

- A notation for expressing the design that seamlessly integrates with a programming language, making it easy to learn and use.
- Formal specifications, expressed as assertions in Floyd-Hoare style [Hoare 1969].
- Specifications checkable at run-time and whose violations may be handled by a system of exceptions.
- Automatic software documentation.

However, DbC is not by itself an example of a formal method as defined above. When we additionally enforce proving that the software components fit their specifications, we are using VDbC. This can be carried out within several environments, which all share the characteristics mentioned above:

- The Java Modeling Language (JML) implements DbC in Java. VDbC can then be carried out using tools like Extended Static Checking (ESC/Java) [Cok 2005] or TACO [Galeotti 2010].
- Perfect Developer [Crocker 2003] is a specification and modeling language and tool which, together with the Escher C Verifier, allows performing VDbC for C and C++ programs.
- Spec# [Barnett 2004] allows VDbC within the C# framework.
- Modern Eiffel [Eiffel 2012] allows it within the Eiffel framework.

4.3 Adaptation

Figure 17 displays the PSP. We assume the engineer will be using an environment like any of those listed at the end of the previous section. This means that a computerized tool (akin to a verifying compiler) is used for

- Checking syntax of formal assertions. These are written in the language employed in the environment (e.g. as Java Boolean expressions, if we were employing JML) which we shall call the carrier language.
- Computing proof obligations (i.e. given code with assertions, to establish the list of conditions that need to be proven to ascertain the correctness of the program).
- Developing proofs in a semi-automatic way.

We now briefly review Figure 17 summarizing the most relevant novelties of PSP:

- After the Design Review phase, there comes a new phase called Formal Specification. It is in this phase that the design is formalized, in the sense that class invariants and pre- and post-conditions to methods are made explicit and formal (in the carrier language).
- The Formal Specification Review has the purpose of detecting errors injected in the formal specification produced in the previous phase. A review script is used in this activity.
- The Formal Specification Compile phase consists of automatically checking the formal syntax of the specification.
- The phase of Code Compile and Proof comes after production and revision of the code. This is the proper formal verification phase, carried out with help of the verifying compiler.

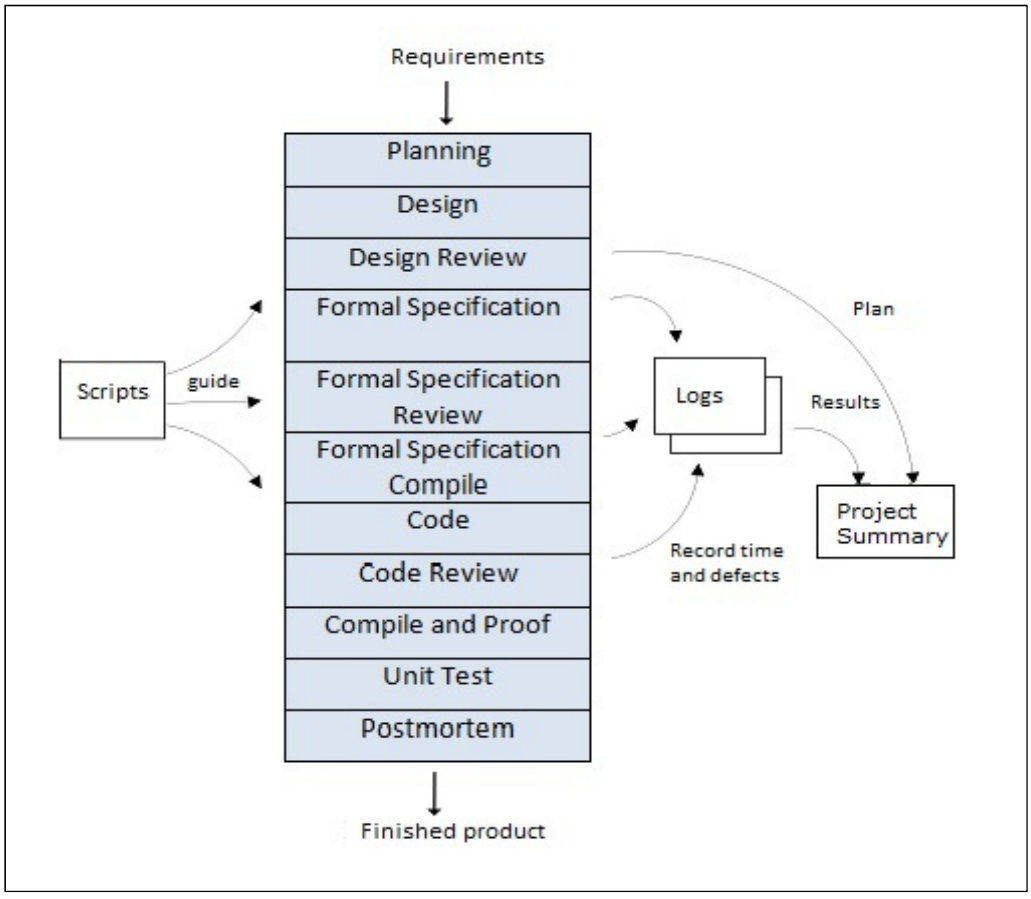


Figure 17: Personal Software Process

In the following subsections we present in detail all the phases of the PSP_{DC}, indicating in each case the activities to be performed and the modifications introduced with respect to the original PSP.

4.4 Planning

The activities of this phase are Program Requirements, Size Estimate, Resource Estimate, Task and Schedule Planning, and Defect Estimate.

Program Requirements is for ensuring a precise understanding of every requirement. This activity is the same as in the ordinary PSP.

Size Estimate involves carrying out a conceptual design (i.e., producing a module [class] structure). Each class is refined into a list of methods and the size of each method is estimated. For this we do as in ordinary PSP (i.e., we use proxies which utilize a categorization of the method according to its size and the functional kind of the corresponding class). Categories of size of methods are very small, small, medium, large, or very large; functional kinds of classes are Calc, Logic, IO, Set-Up, and Text. Thus, using the structure of classes, the number of methods in each class and the category of the class, we arrive at an estimation of the LOCs of the program.

Now, using (Verified) Design by Contract requires us to formally write down the pre- and post-conditions of each method and the invariant of each class. And we do not yet possess methods akin to the above-mentioned proxies for estimating the size of these formal specifications. Therefore, we are not in a position to produce such estimation. However, it can be argued that VDbC modifies the process by which we arrive at design and code, but not the size of the final program in LOCs, at least if we consider the latter to be the executable code that is necessary to achieve the required functionality. In this sense, formal specifications could be treated as formal comments to the code (i.e., not to be counted into the size of the program).

Resource Estimate estimates the amount of time needed to develop the program. For this, the method PROBE is used, which employs historical records and linear regression for producing the new estimation, as well as for measuring and improving the precision of the estimations. In our adaptation, the activity remains conceptually the same, but will employ records associated to the new phases incorporated into PSP_{DC}. Therefore, once sufficient time data has been gathered, we shall be able to estimate the time to be employed in formal specification, as well as in program proof.

Task and Schedule Planning is for long-term projects. These are subdivided into tasks and the time is estimated for each task. This is unchanged in PSP_{DC}.

Defect Estimate Base is for estimating the number of defects injected and removed at each phase. Historical records, as well as the estimated size of the program, are utilized for performing this estimation. In PSP_{DC} we must keep new records in order to estimate the defects removed and injected at each new phase.

4.5 Design

During Design, we define the data structures, classes and methods, interfaces, components, and the interactions among all of them. Formal specification of methods and of invariants of classes

could be carried out within the Design phase. This, however, presents the disadvantage of not allowing us to keep records of the time employed specifically in Design, as well as in Formal Specification. Instead, we would just record a likely significant increase in Design time. Therefore we prefer to separate the phase of Formal Specification.

4.6 Design Review

This is the same as in ordinary PSP.

4.7 Formal Specification

This phase must be performed after Design Review. The reason for this is that reviews are very effective in detecting defects injected during design, and we want these to be discovered as early as possible.

In this phase we start to use the computerized environment supporting VDbC. We propose to carry out two activities within this phase, namely Construction and Specification. The activity of Construction consists of preparing the computerized environment and defining within it each class with its method headers. This could have been done during Design, in which case we would, of course, omit it here. The choice is a valid personal one. The second activity is Specification, in which we write down in the carrier language the pre- and post-conditions of each method as well as the class invariant. Notice that, within the present approach, the use of formal methods begins once the design has been completed. It consists of the formal specification of the produced design and the formal proof that the final code is correct with respect to this specification.

4.8 Formal Specification Review

Using a formal language for specifying conditions is not a trivial task and both syntactic and semantic defects can be injected. To avoid the propagation of these errors to further stages and thereby increasing the cost of correction, we propose to carry out a phase of Formal Specification Review.

The script that corresponds to this phase contains the activities of Review, Correction, and Checking. The Review activity consists in inspecting the sentences of the specification using a checklist. In the activity of Correction all defects detected during Review are removed. Finally, Checking consists of looking over the corrections to verify their adequacy.

4.9 Formal Specification Compile

Any computerized tool supporting VDbC will be able to compile the formal specification. Since this allows an early detection of errors, we consider it valuable to explicitly introduce this phase into PSP_{DC}. In particular, it is worthwhile to detect all possible errors in the formal specifications before any coding is carried out. A further reason to isolate the compilation of the formal specification is that it allows recording the time spent in this specific activity.

4.10 Code

Just as in ordinary PSP, this phase consists of translating the design into a specific programming language.

4.11 Code Review

This phase does not differ from the corresponding one in ordinary PSP.

4.12 Compile and Proof

The phase Code Compile of PSP is modified in PSP_{DC} in order to provide, besides the compiled code, evidence of its correctness with respect to the formal specification (i.e., its formal proof). As already said, we here use the computerized tool (verifying compiler) which compiles the code, derives proof obligations, and helps to carry out the proofs themselves.

4.13 Unit Test

This phase is the same as in ordinary PSP. We consider it relevant for detecting mismatches with respect to the original, informal requirements to the program. These defects can arise in several points during the development, particularly as conceptual or semantic errors of the formal specifications. The test cases to be executed must therefore be designed right after the requirements are established, during the phase of Planning.

4.14 Post-Mortem

This is the same as in ordinary PSP. Several modifications have to be made to the infrastructure supporting the new process. For instance, all new phases must be included into the support tool to keep control of the time spent at each phase as well as to record defects injected, detected, and removed at each phase. Our intention in this paper is to present the changes in the process in order to incorporate VDbC. The adaptation of the supporting tools, scripts, and training courses is a matter for a separate work.

4.15 Conclusions and Future Work

We have presented PSP_{DC}, a combination of PSP with Verified Design by Contract (VDbC), with the aim of developing better quality products.

In summary we propose to supplement the design with formal specifications of the pre- and post-conditions of methods as well as class invariants. This gives rise to three new phases that come after the Design phase, namely Formal Specification, Formal Specification Review, and Formal Specification Compile. We also propose to verify the logical correctness of the code by using an appropriate tool, which we call a *verifying compiler*. This motivates the modification of the Compile phase, originating the new Compile and Proof phase, which provides evidence of the correctness of the code with respect to the formal specification.

The process can be carried out within any of several available environments for VDbC.

By definition, in Design by Contract (and thereby, also of VDbC) the specification language is seamlessly integrated with the programming language, either because they coincide or because the specification language is a smooth extension of the programming language. As a consequence, the conditions making up the various specifications are simple to learn and understand Boolean expressions. We believe that this makes the approach easier to learn and use than the ones in other proposals, like Babar and Suzumori [Babar 2005, Suzumori 2003]. Nonetheless, the main

difficulty associated with the whole method resides in developing a competence in carrying out the formal proofs of the written code. This is, of course, common to any approach based on formal methods. Experience shows, however, that the available tools are generally of great help in this matter. There are reports of cases in which the tools have generated the proof obligations and discharged up to 90% of the proofs in an automatic manner [Abrial 2006].

We conclude that it is possible in principle to define a new process that integrates the advantages of both PSP and Formal Methods, particularly VDbC. Our future work consists of completing the adaptation of PSP by writing down in detail the scripts to be used in all phases, adapt the logs, specify and carry out the modifications to the support tool, and modify or define interesting metrics.

We must evaluate the PSP_{DC} in actual practice by carrying out measurements in case studies. The fundamental aspect to be measured in our evaluation is the quality of the product, expressed in the amount of defects injected and removed at the various stages of development. We are also interested in measures of the total cost of the development.

4.16 Author Biographies

Silvana Moreno
School of Engineering, Universidad de la República

Silvana Moreno is a teaching and research assistant at the Engineering School at the Universidad de la República (UdelaR). She is also a member of the Software Engineering Research Group (GrIS) at the Instituto de Computación (InCo.). Moreno holds an Engineer title in computer science from UdelaR and is currently enrolled in their Master of Science program in computer science.

Álvaro Tasistro
School of Engineering, Universidad ORT Uruguay

Álvaro Tasistro is a professor at the engineering school of the Universidad ORT Uruguay. He is a chair of Theoretical Computer Science and is also the director of studies for the Master of Engineering Sciences program at the university. He holds a PhD from Chalmers University in Gothenburg, Sweden. He has several articles published in international conferences, journals, and books. His main research topics are type theory and formal methods.

4.17 References/Bibliography

[Abrial 2006]
Abrial, Jean-Raymond. "Formal Methods in Industry*:* Achievements, Problems, Future,*"* 761–768. *28th International Conference on Software Engineering (ICSE'06*). Shanghai, Republic of PRC, May 2006. ACM Press, 2006.

[Babar 2005]
Babar, Abdul & Potter, John. "Adapting the Personal Software Process (PSP) to Formal Methods," 192-201. *Australasian Software Engineering Conference (ASWEC'05*), Brisbane, Australia, March-April, 2005. IEEE, 2005.

[Barnett 2004]
Barnett, Mike; Rustan, K.; Leino, M.; & Schulte, Wolfram. "The Spec# Programming System: An Overview" *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), Lecture Notes in Computer Science 3362*, pages 49–69. Springer, 2004.

[Cok 2005]
Cok, David & Kiniry, Joseph. "ESC/Java2: Uniting ESC/Java and JML." *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS 2004). Lecture Notes in Computer Science 3362*, pp. 108–228. Springer-Verlag, 2005.

[Crocker 2003]
Crocker David, *Perfect Developer: A Tool for Object-Oriented Formal Specification and Refinement*, 2003.

[Eiffel 2012]
Eiffel. Definition of "Modern Eiffel." http://tecomp.sourceforge.net/index.php?file=doc/papers/lang/modern_eiffel.txt. Retrieved August 16, 2012.

[Frege 1967]
Frege, G. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens.* Halle a. S.: Louis Nebert, 1879. Translated as *Concept Script, a formal language of pure thought modelled upon that of arithmetic*, by S. Bauer-Mengelberg. Edited by J. vanHeijenoort, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, Cambridge, MA: Harvard University Press, 1967.

[Galeotti 2010]
Galeotti Juan; Rosner, Nicolás; Pombo, López; &. Frias, Marcelo F. "Analysis of invariants for efficient bounded verification," 25-36. *Proceed*ings *of the Nineteenth International Symposium on Software Testing and Analysis* (ISSTA 2010), Trento, Italy, July 2010. Schloss-Dagstuhl, 2010.

[Hoare 1969]
Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." *Communications of ACM 12*, 10 (October 1969): 576-580.

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005.

[Humphrey 2006]
Humphrey, Watts S. *TSP: Coaching Development Teams.* Addison-Wesley, 2006. http://www.sei.cmu.edu/library/abstracts/books/201731134.cfm

[Meyer 1992]
Meyer, Bertrand. "Applying Design by Contract." *IEEE Computer* 25, 10 (October 1992): 40-51.

[Schwalbe 2007]
Schwalbe, Kathy. *Information Technology Project Management*, 5th ed. Course Technology, 2007.

[Suzumori 2003]
Suzumori, Hisayuki; Kaiya, Haruhiko; & Kaijiri, Kenji. "VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development," *Proceedings of the 27th Annual International Computer Software and Applications Conference*. Hong Kong, PRC, September–October 2003. IEEE Computer Society, 2003.

5 Experience Report: Applying and Introducing TSP to Electronic Design Automation

Elias Fallon and Lee Gazlay, Cadence Design Systems, Inc.

5.1 Introduction

This paper will describe our experiences in introducing the Team Software Process (TSP) to a software research and development (R&D) division of Cadence Design Systems. We will describe the Electronic Design Automation (EDA) industry and some of the challenges that led us to pilot and start adoption of TSP. We will also examine the specific results we have achieved from teaching PSP Fundamentals to Cadence engineers and characterize the similarities and differences to the historical Personal Software Process (PSP) results. Finally, we will summarize some of the successes and challenges we have had in using TSP on the day-to-day work.

In April 2011, Cadence Design Systems officially kicked off pilot projects using the TSP. We are interested primarily in using TSP to help us develop software at a significantly higher quality while maintaining a high productivity. This initiative started in the Custom IC and Simulation division of Cadence, with roughly a quarter of Cadence's 2400 engineers in R&D. We currently have two teams with more than a year of experience with TSP, plans to have a total of ten teams trained and using TSP by the end of 2012, and further expansion planned in 2013. Cadence is a leading company in EDA, which gives us an interesting perspective and motivation for engineering quality into our software products.

5.2 Electronic Design Automation

EDA companies serve the semiconductor and electronics industry with software tools to enable their design and manufacturing needs. Our industry is fueled by Moore's Law [Wikipedia 2012], addressing the continuous exponential increase in performance and capacity per price being delivered by our semiconductor technology. What is less well known is that all of these "newer/bigger/faster" chips and electronics have been created each year by a relatively stable number of engineers. While the performance and capacity of electronics keeps doubling every 18–24 months, the number of engineers doing these designs has increased only marginally. To facilitate this technological growth, the EDA industry has continued to deliver higher levels of productivity-enabling software. EDA companies serve an industry that is highly quality conscious due to the cost of manufacturing and volumes required. We sell software that helps chip designers estimate the yield through their manufacturing process of a given chip design and offer suggestions on how to improve that yield. And yet, applying similar techniques to our own software development processes has not been our primary objective. Due to the constantly changing technology our customers are working with, we often have to develop new products and features very quickly in close partnership with our customers to enable the next generation of designs. This requires us to deliver significant new functionality quickly, traditionally at the expense of quality, and has given our industry a reputation for innovation but only just-good-enough quality [Ben-Yaacov 2001]. With TSP, we believe we can transform our company to deliver that same new functionality, just as fast, with true production quality. The advantages would be significant for both our customers and for us.

5.3 Why We Decided to Pilot TSP

The problem of developing interesting and innovative software with high quality and on a predictable schedule is well known in the industry. Clearly, we are not the first to make the analogy between electronics design, implementation, and manufacture, and software [Humphrey 2009], however, the analogy bears repeating. The majority of Cadence developers still work with languages, tools, and flows for developing software that hasn't changed appreciably in 20 years. Continuing to innovate in our fast-paced industry, while maintaining the millions of lines of code in our many products, is a significant challenge. Finding ways of improving our developer productivity and improving the overall quality experience of our products for our customers is critical for our continued competitiveness.

Cadence, and specifically the Custom IC and Simulation division, has been focused on improving the quality delivered to customers in recent years. The initial focus has been on incrementally improving the software development processes already in place. Table Table 14 shows the actions taken between the 6.1.45 and 6.1.4 releases of our software, and the measured increases in improved testing/validation/quality activities.

Table 14: Comparison of Quality Measures for 6.1.5 vs. 6.1.4

Quality Measure	**6.1.5 vs. 6.1.4**
Automated Tests	22% increase in quick tests 30% increase in regression (system) tests
Performance Testing	82% increase in number of automated performance benchmarks
Customer Acceptance Testing (CAT)	3 Active vs. None
Licensing Testing	93% increase in automated license tests.
Coverity (static code analysis)	0 Defects at release 6.1.5 vs. 314 for 6.1.4
Purify / Valgrind (dynamic code analysis)	0 Defects at release 6.1.5 vs. 382 for 6.1.4

The size and effort of both of these releases were significant, involving hundreds of engineers for 12–18 months of development/release activities. From these improved metrics, we expected to see significantly improved quality visible to the customer when the software was delivered. Unfortunately, a view of the software from both the perspective of number of incoming bug reports, (called *CCRs* in our internal bug tracking system) and crash rates measured at key customer sites (percent of software starts that end in a crash), showed the quality to be roughly the same as the previous release. Deeper analysis of the incoming CCRs showed that most problems were reported on the newly developed functionality, while the older functionality was more stable/improved compared to previously. Discussions with our customers also showed that their overall perception was that the new release was higher quality than before. So these were still successful releases (now being used in production by hundreds of customers). But we suffered all of the classic issues associated with too many latent defects. Much of our engineering resources had to go to fixing the critical defects and delivering many updates to the software to the customers to stabilize on a working system. This led us to the conclusion that we had to fundamentally change how we develop software and find techniques that allowed our developers to manage quality better within their own process, rather than continue the spiral of ever-

increasing testing. Toward this end, we began the TSP pilot project, followed by a larger rollout program now in progress.

We are planning a four-year rollout program to the roughly 40 teams and 450 engineers composing the Custom IC and Simulation R&D division. The rollout began with the two-team pilot project and some initial training in 2011. It continues this year with a target of introducing TSP in the daily work of an additional 10 teams. In 2012 and 2013, we plan to add roughly 15 teams each year. Beyond that, we are beginning to coordinate informational sessions and pilot projects for other divisions at Cadence. One of the primary challenges we see in driving adoption is finding enough experienced engineers within Cadence who are willing and able to become TSP Coaches to help sustain and grow the program.

5.4 PSP Results

To date we have trained 66 Cadence engineers on PSP Fundamentals. The vast majority of the engineers have done the assignments in C/C++, our standard programming language for our applications, although a few have used other languages as well (Tcl, Perl, and PHP). The results obtained from examining all of the data collected from those 264 programs are consistent with the historical PSP results [Burton 2006].

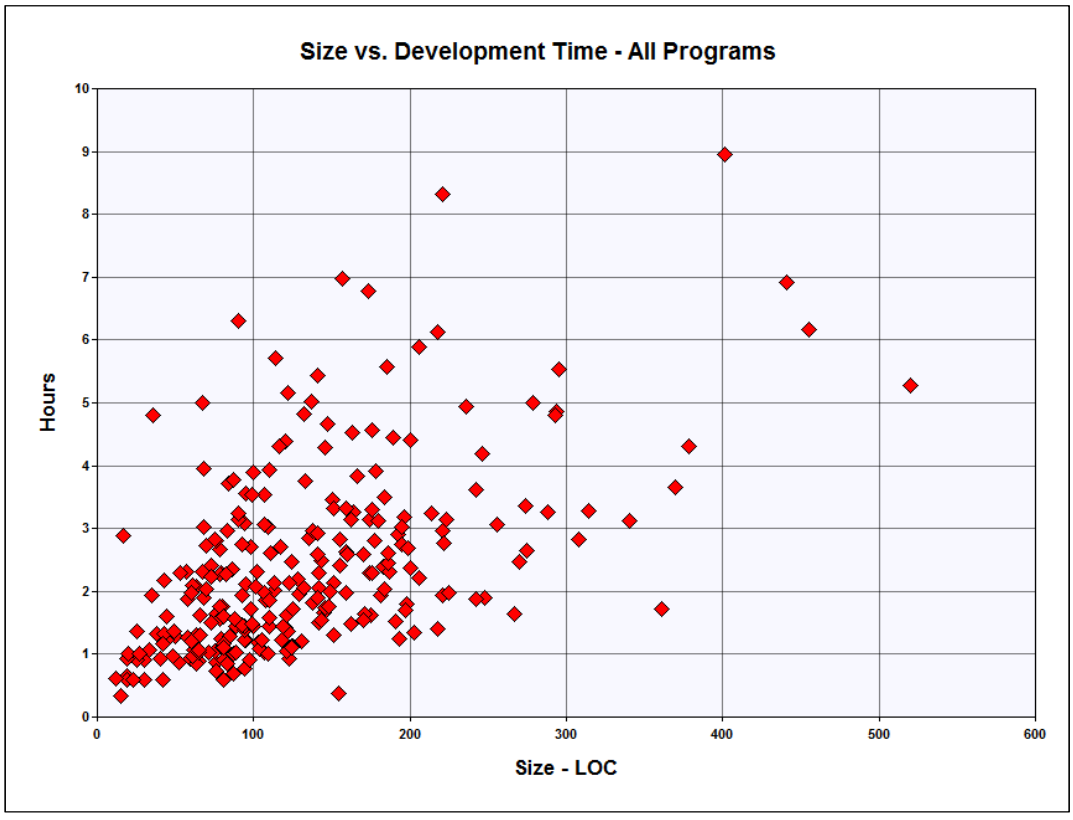


Figure 18: Size Versus Development Time for All PSP Fundamental Programs Written at Cadence

We see a correlation between program size (added and modified LOC) and development time in PSP Fundamentals. We also clearly see a wide variation due to individual productivity differences and the process changes that the engineers go through in their four programs.

The average total defects found in our classes, ranging from 50–80 defects/KLOC, tend to be lower than that reported in the SEI's literature [Burton 2006] for Programs 1–4, where the typical range for C/C++ is 80–120 Defects/KLOC. Our guesses as to the causes of this difference are experience of our developers; some use of advanced IDEs with syntax checking/fixing; and pre-existing mature processes, such as significant design documentation, followed even in Program 1.

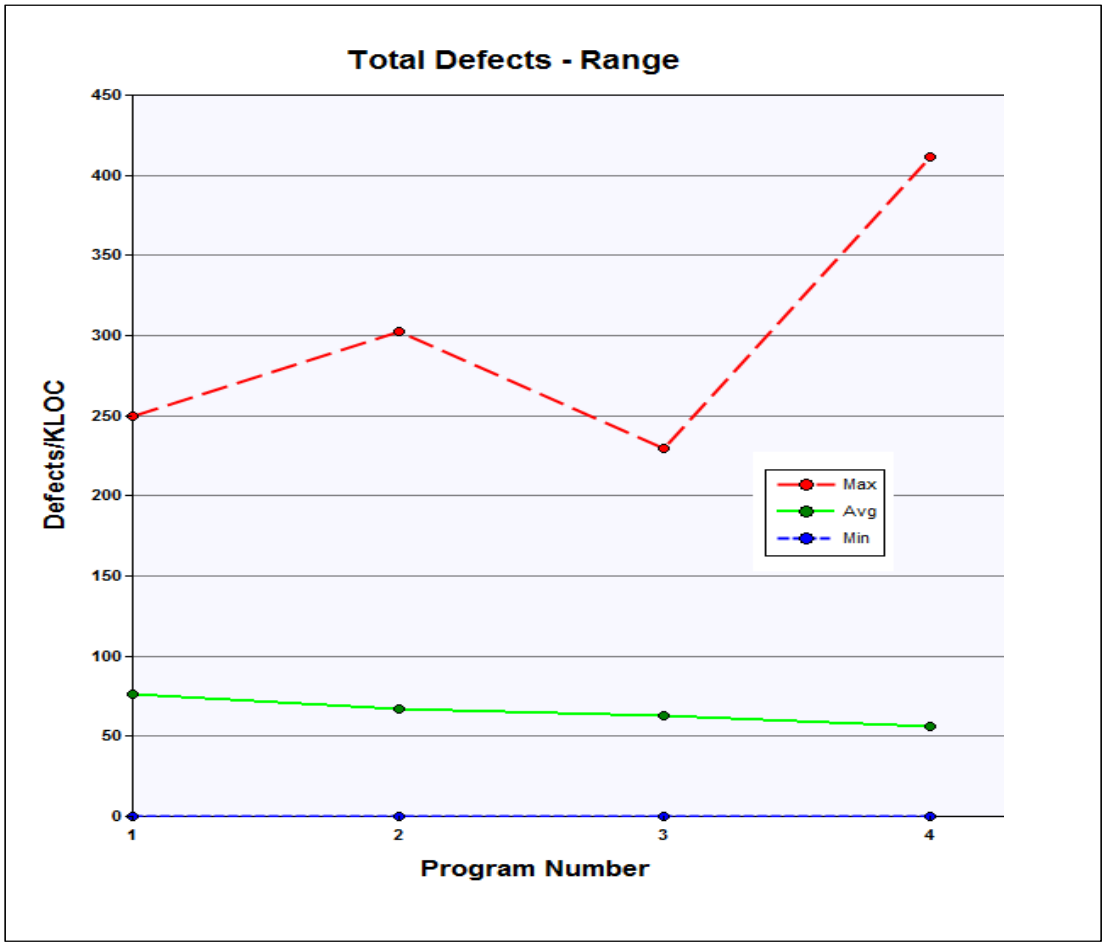


Figure 19: Total Defects Injected in PSP Fundamentals Programs

However, even with this lower baseline of total defects, we do see measurable quality improvements during the class as the students proceed from PSP0 through PSP1 and PSP2 processes. In the chart below, we see the defect removal rates for the four programs, with the introduction of Design and Code Reviews in Programs 3 and 4. The Code Review Defect removal rate averages slightly more than double the test defect removal rate.

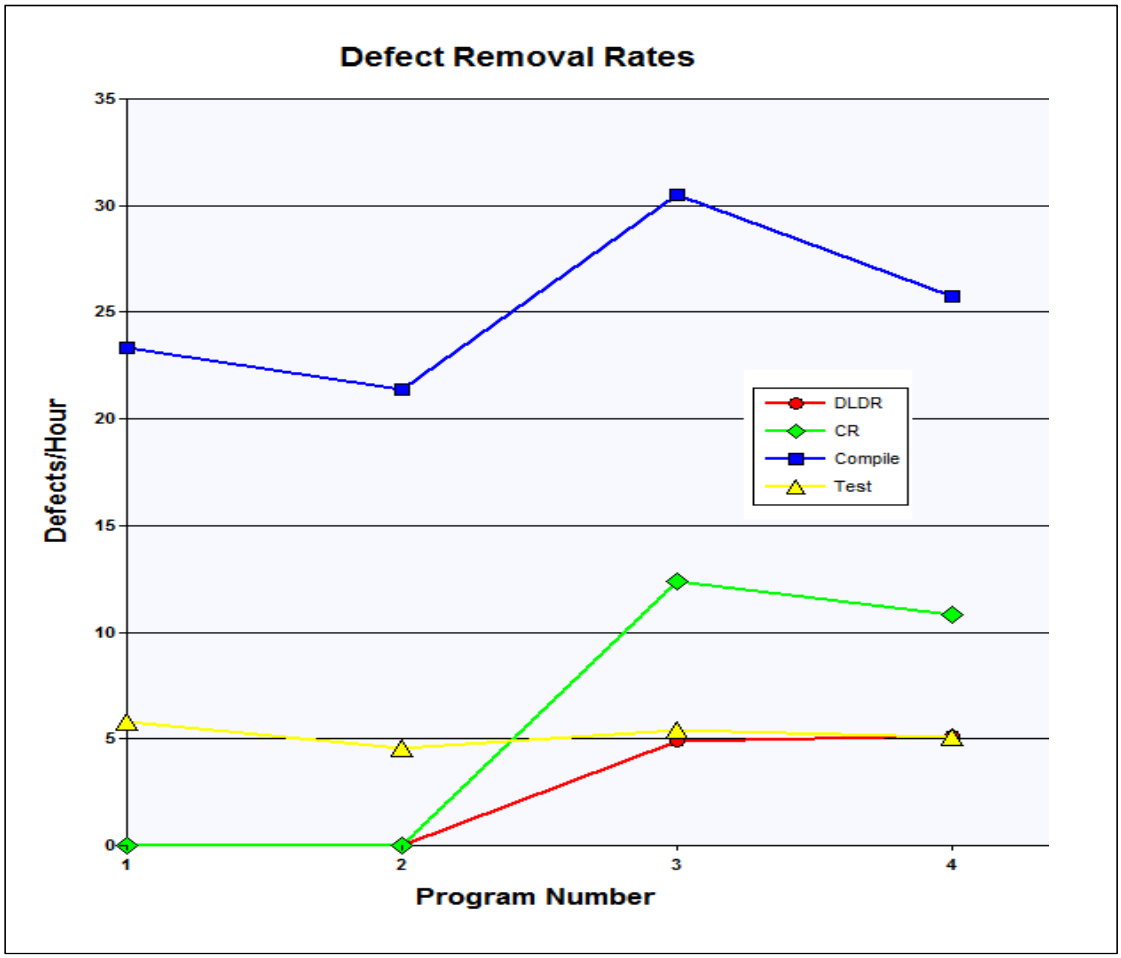


Figure 20: Defect Removal Rates for PSP Fundamentals Programs

We also see the very nice trend of declining total quality costs from Program 1 through Program 4, from above 30% down to almost 25%. Figure 21 shows the impact of the structured reviews, with unit test defect densities halving after their introduction. Figure 22 and Figure 23 demonstrate the cost of quality and the contribution of appraisals to this cost. This data, combined with Figure 24 data, shows productivity increasing slightly in Programs 3 and 4 compared to Program 2. These results demonstrate that our engineers were able to add whole new steps to the PSP development process and actually go faster than before, due to spending less time fixing defects in compile and test.

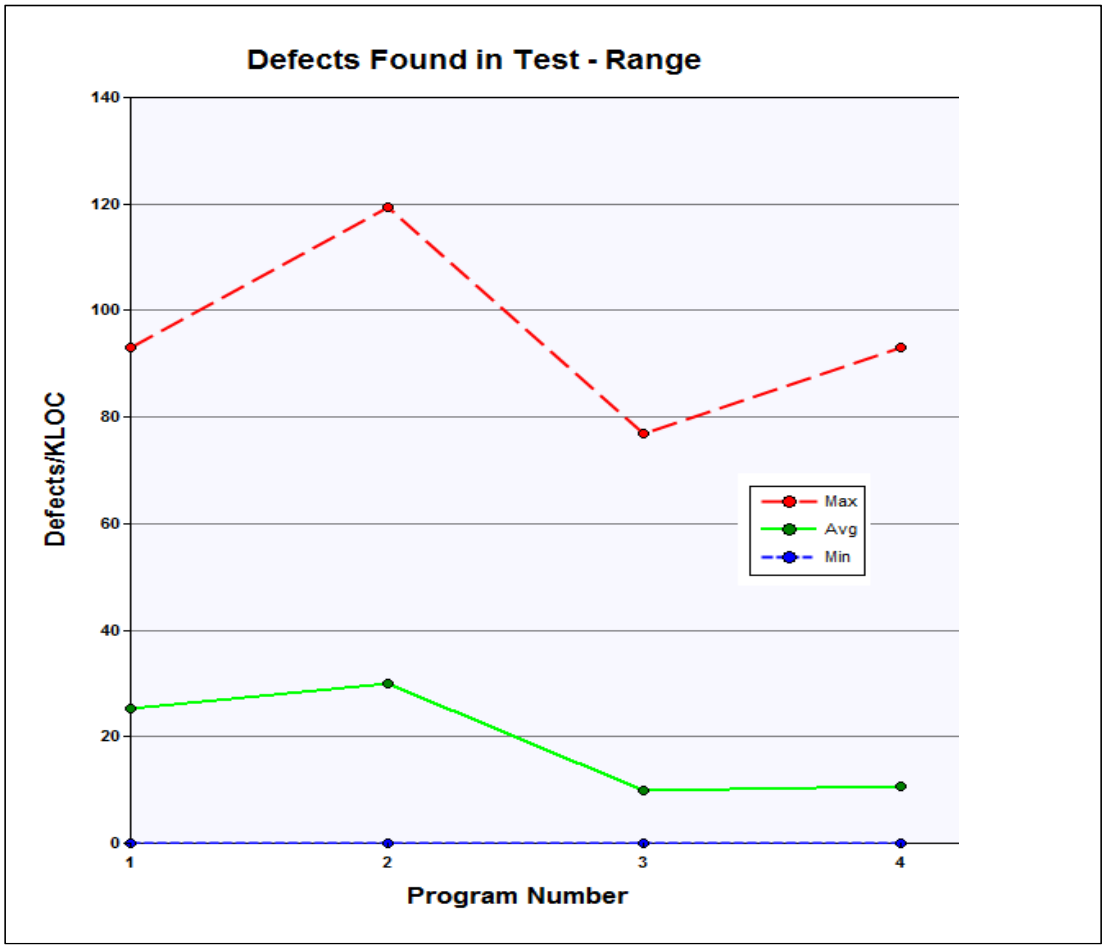


Figure 21: Test Defect Density Across Programs

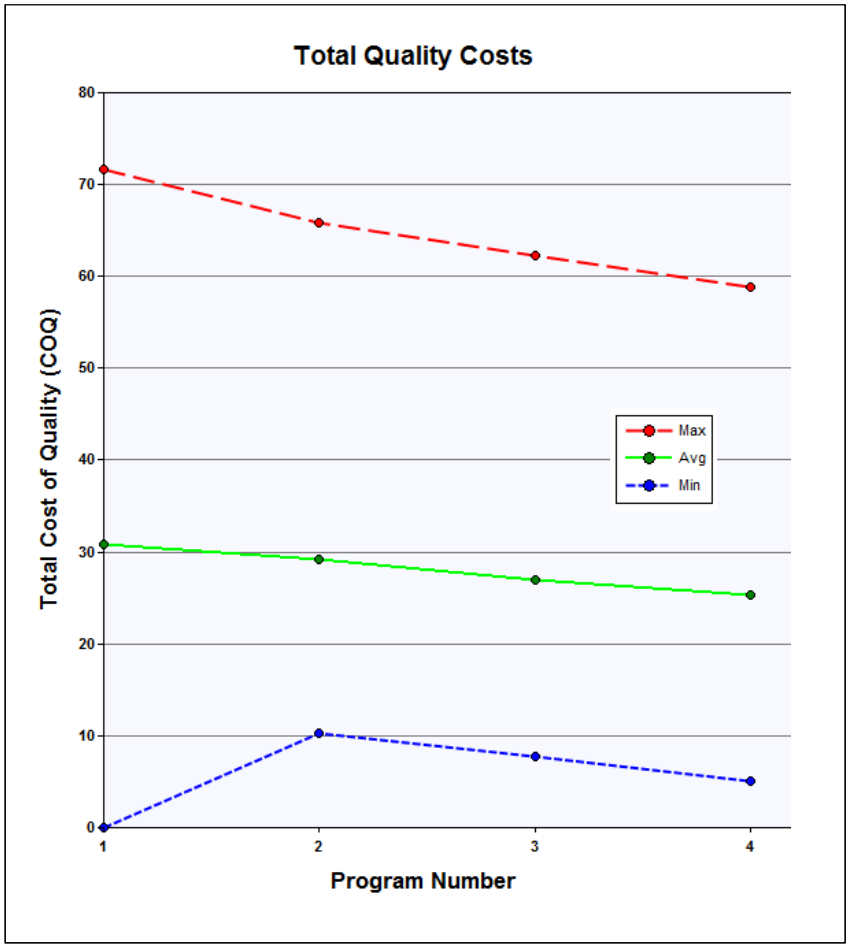


Figure 22: Total Quality Costs

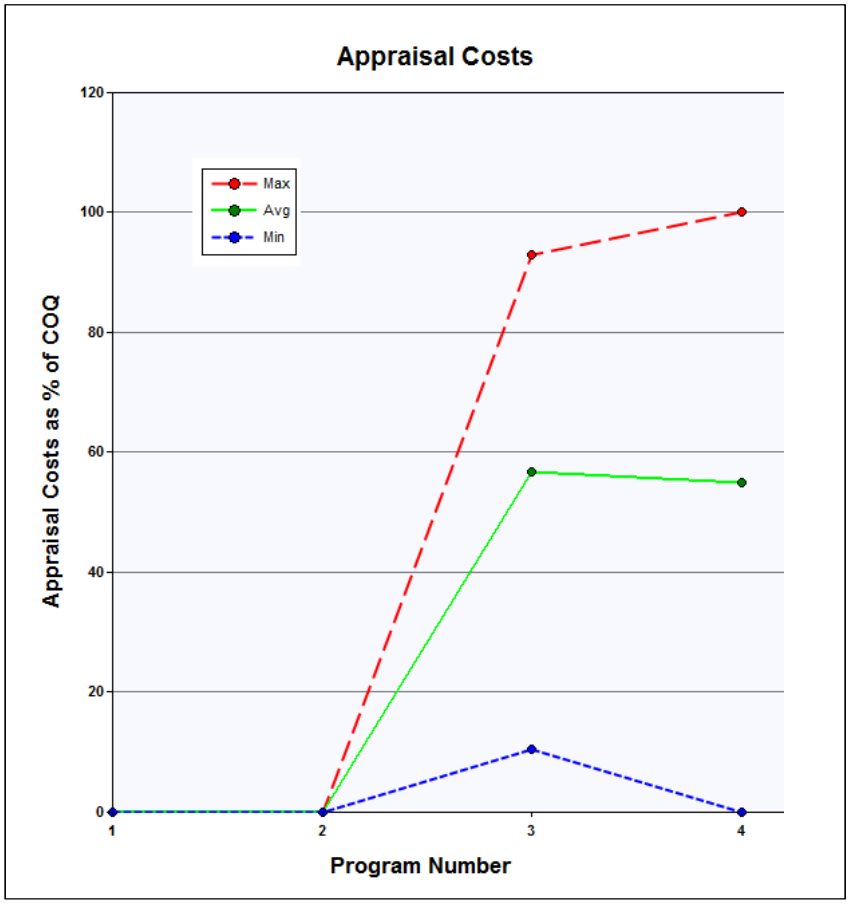


Figure 23: Appraisal Costs for PSP Fundamentals Programs

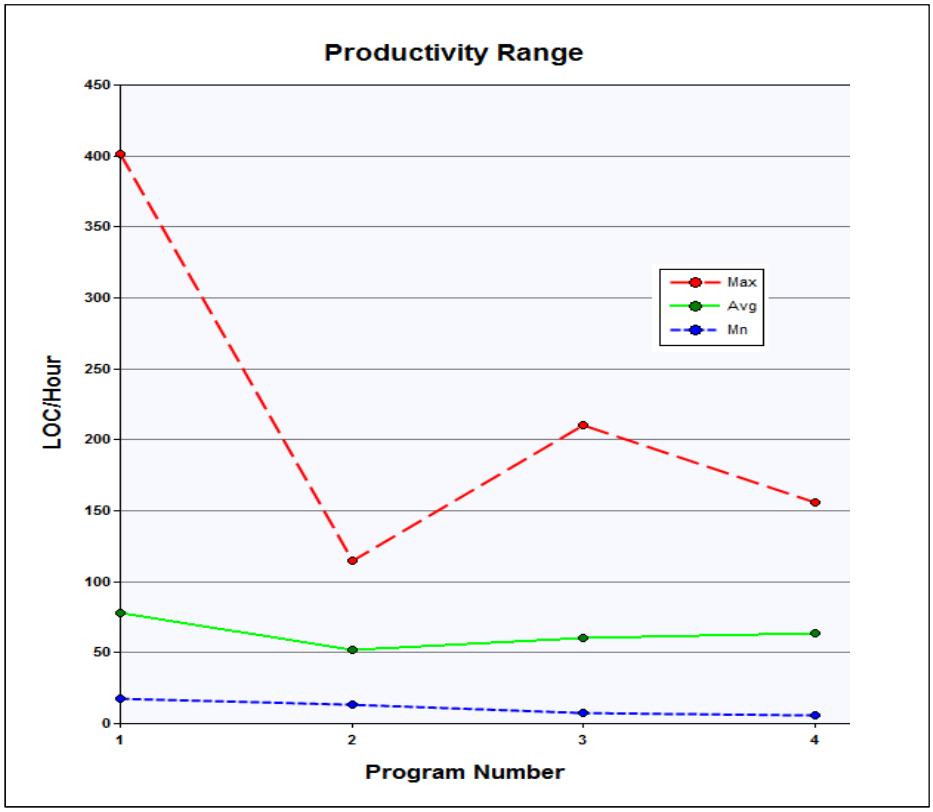


Figure 24: Productivity for PSP Fundamentals Programs

The results of PSP Fundamentals at Cadence, confirming the academic research on PSP, have helped the development teams see the value in measuring their work and using the quantitative results to improve their own performance. The collective results show the effectiveness and efficiency of using PSP as well as some of its best practices, such as checklist-based reviews. In the traditional TSP structure, we have subsequently moved each team on to a TSP Launch.

5.5 TSP Results

We now have two teams, each with several TSP development cycles completed, and another five teams that are on their first or second TSP development cycle. These teams range from 6 engineers in the same office, to 23 engineers spread across four offices and nine time zones. We have seen good results so far in terms of quality and are able to show review and inspection processes that are much more efficient and effective than testing. We have had less success with planning and estimating the work. Our teams have all faced challenges with TSP planning, given that each team has some combination of maintenance and new development work, with short turnaround times required on the maintenance work.

5.5.1 TSP Quality Results

The following three charts (Figure 25, Figure 26, and Figure 27) show the defect removal profiles of a single team across three development cycles.

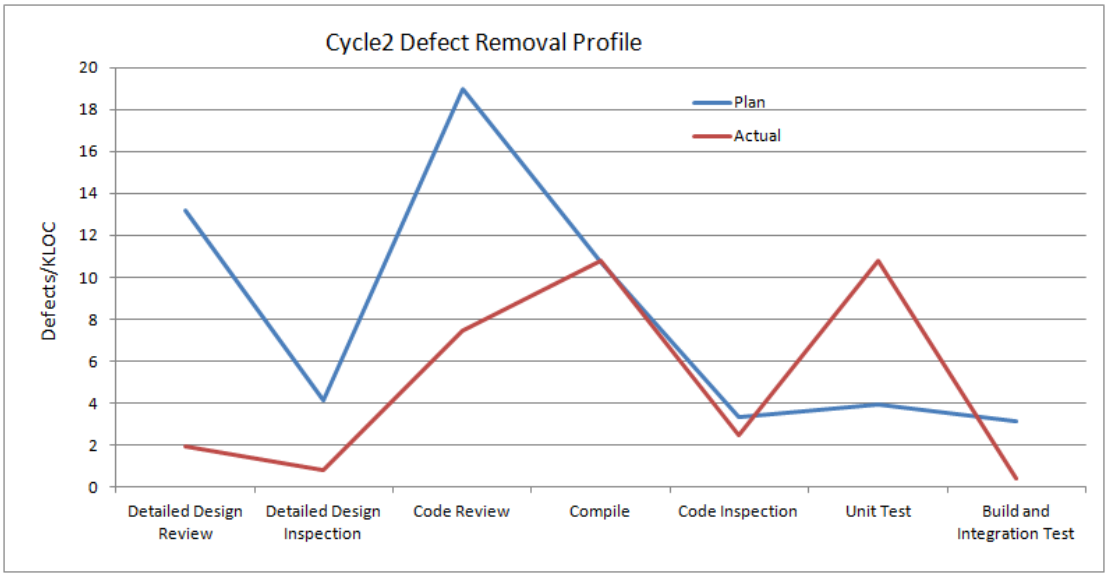


Figure 25: Defect Removal Profile (Defects/KLOC Removed in Each Phase) for a Team in Its Second Cycle

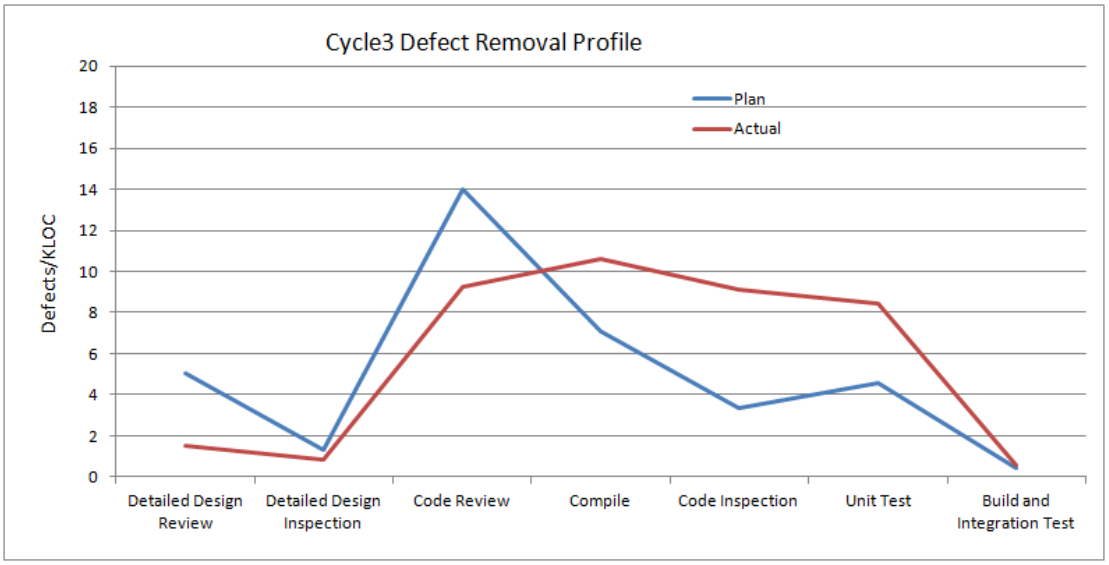


Figure 26: Defect Removal Profile (Defects/KLOC Removed in Each Phase) for the Same Team in Its Third Cycle with a More Formal Code Inspection Process

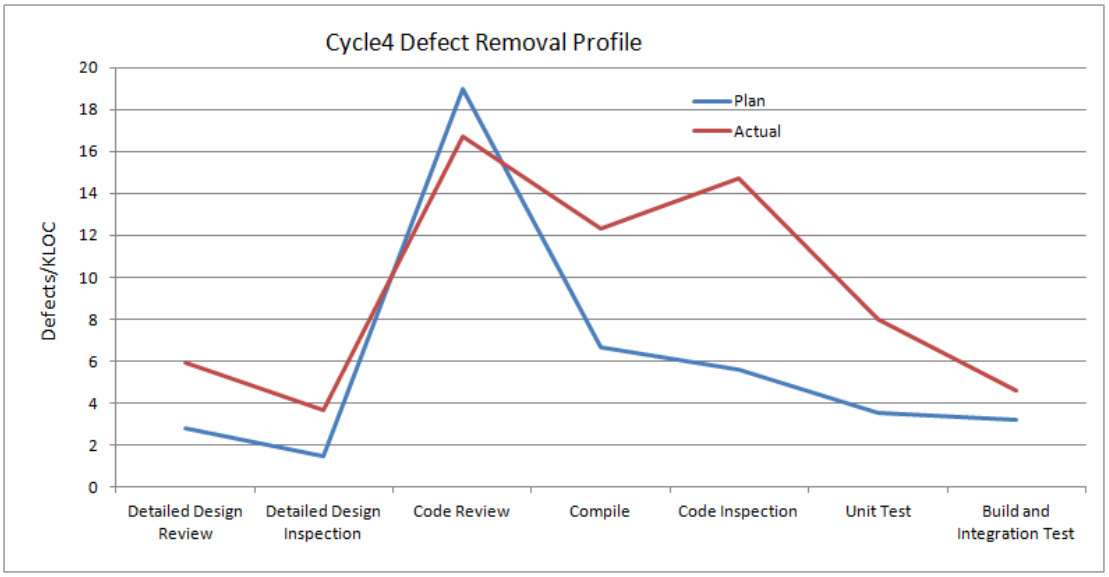


Figure 27: Defect Removal Profile (Defects/KLOC Removed in Each Phase) for the Same Team in Its Fourth Cycle

In this team's Cycle 2, the Code Inspection was more of an informal pair walkthrough process. In Cycle 3, a formal checklist and tool-based process was introduced and was continued in Cycle 4. While we can see that more defects were still detected in test than planned, a much higher density of defects was detected in Code Inspection, compared to the previous cycles. Much of this consisted of legacy defects or opportunities for refactoring or code improvement that would previously have been left for future works and probably never addressed. While this data focuses on a single team, we have observed similar evolutions across other teams. The TSP process and data allows the teams to successfully adopt quality practices they might not have spent the time on previously.

With all of this data we are also able to see the cost of quality inside the developers' process across a number of teams.

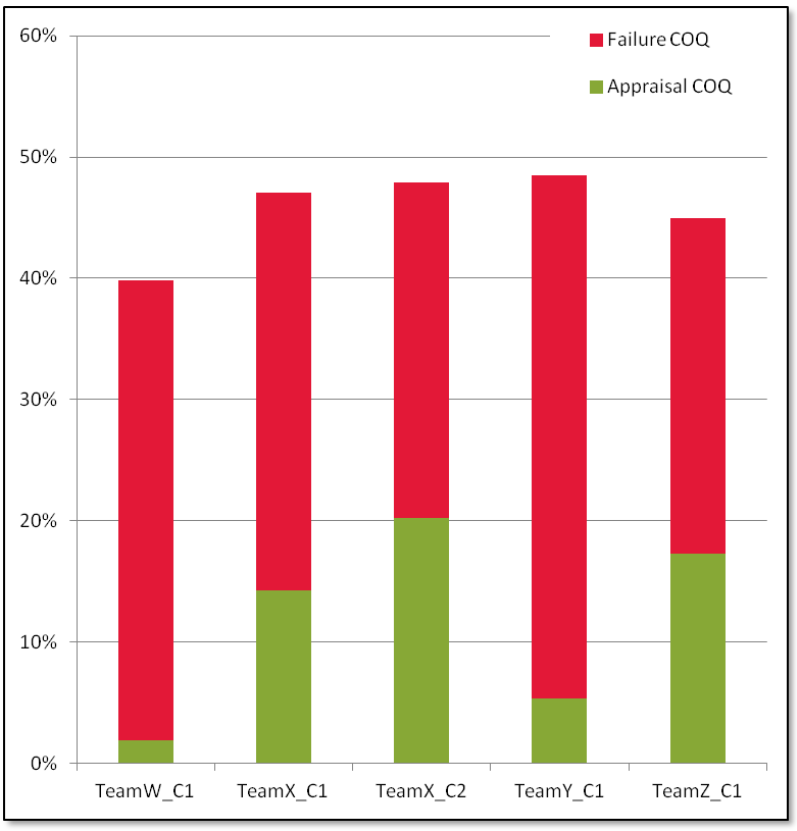


Figure 28: Cost of Quality Across Four Teams Inside the Developer's Process

Since this Cost of Quality figure only measures inside TSP processes, it is really only a lower bound on the actual figure. But the ability to measure COQ and track our improvements have energized our engineers to drive the process improvements needed to develop high-quality software as efficiently and effectively as possible. We consider the overall quality improvements we have seen on TSP teams to be a major success for the program.

5.5.2 TSP Planning Results

Overall we have been successful in using the TSP framework to create and track our plans. For many of our teams however, we are still ending our development cycles at roughly 60%–70% of our planned earned value (EV). Anecdotally, our development teams report this as being similar to their previous planning results. A good portion of our team's planned work for each development cycle has been spent on bug fixes or small enhancements (dominated more by modifications than by new code). For our teams this has been 40%–75% of the work they have planned for a development cycle. Typically this has involved both known and unknown bug fixes. We have attempted to include the unknown by measuring historical rates of incoming critical bug fixes that will need to be addressed in any given time period, but aren't known at the start of that time period, and including placeholder tasks in our TSP plans for each cycle. For both the known and unknown fixes, however, we have had trouble converging our planning sufficiently to predict our work. Figure 29 shows the planned versus actual time for "modification-dominated" work in the third development cycle for a particular development team.

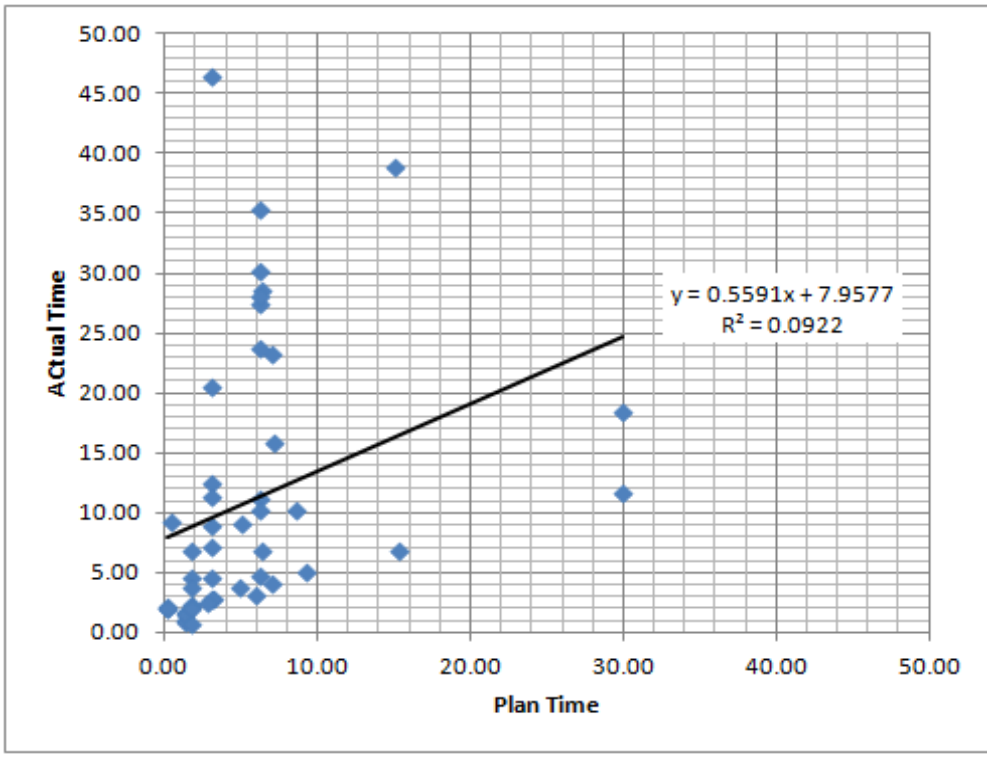


Figure 29: Plan Time vs. Actual Time Team X, Cycle 3

By the third cycle we were hoping to have seen significant improvements in planning convergence. Further investigation showed that the primary issue for planning was in size estimation.

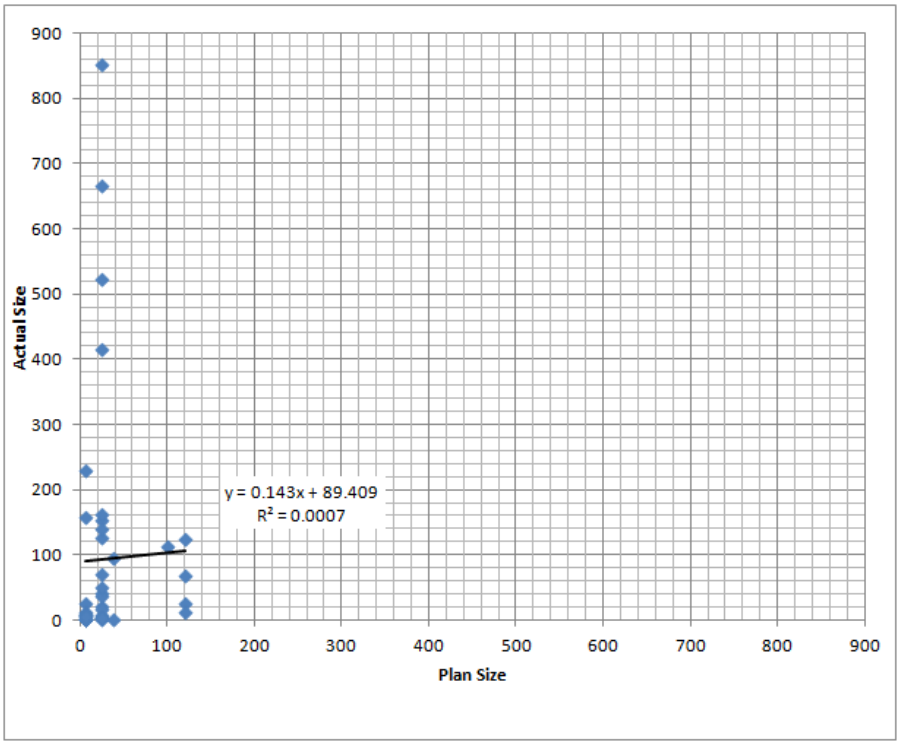


Figure 30: Plan Size Versus Actual Size for the Same Development Items Shown in Figure 28

Clearly, we weren't having much success estimating the size of these modification-dominated works. However, even beyond that, we found that measuring the size of added and modified LOC really didn't correlate to development time either. If we focus on the changes that involved less than 100 added and modified LOC, we see the following:

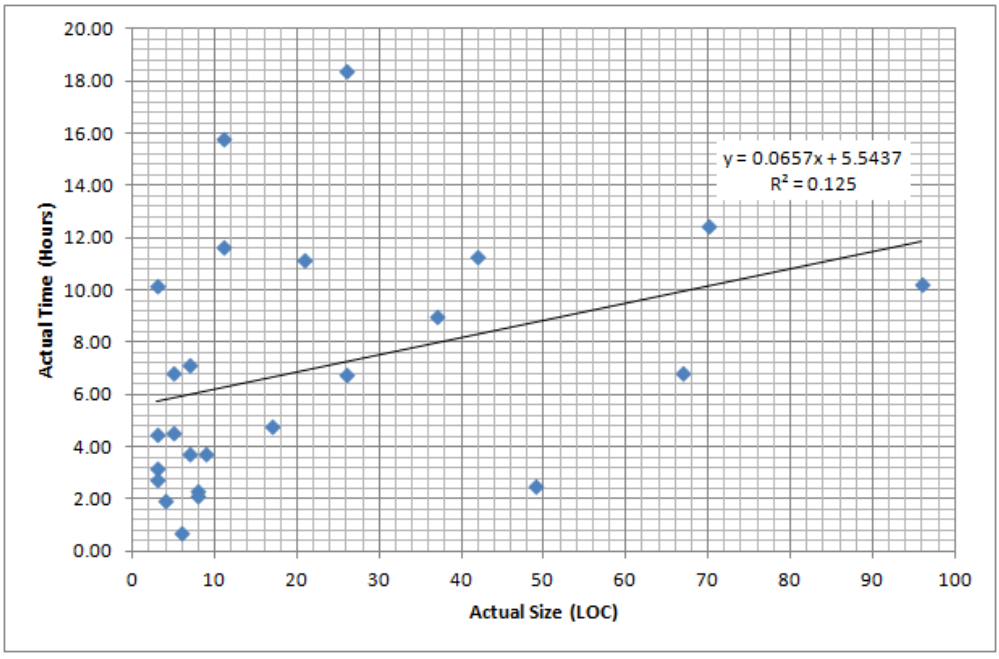


Figure 31: Actual Size vs. Actual Time for Modification-Dominated Tasks with Less than 100 A&M LOC

Here we essentially see no correlation between size and time, indicating this is not a good size measure for this type of work. We intend to address this problem in two ways:

1. (Long Term) Continue researching possible size proxies that do correlate with work, such as
   a. number of functions/modules touched
   b. number of places modified functions are called from
   c. cyclomatic complexity of modified functions
2. (Short Term) Create proxy tables to go from pure "proxy size" (Small, Medium, Large) directly to effort, based on historical average and standard deviation of actual time for similar types of work.

The planning and estimation remains one of the challenging areas of TSP adoption on the teams.

5.6 Summary

We are continuing an aggressive rollout schedule of TSP across one entire division at Cadence, and we are discussing starting pilots with the other organizations. In this we are facing many of the same challenges as other large organizations in terms of the time and effort, and additional bandwidth needed to roll out these changes to the entire organization.

We have described the challenges we've experienced in trying to make traditional test and tool-based quality improvements and strived to show the meaningful impact on our product quality that has led us to TSP. We have described the successes we have seen using PSP Fundamentals to show the potential of PSP/TSP to improve quality. In addition we have shown some of the progress we have made in improving quality practices using TSP primarily around adoption of structured reviews and inspections. Finally, we have described the challenge of adapting TSP planning practices to our modification-dominated work.

5.7 Acknowledgements

We would like to thank the two TSP pilot teams, the Virtuoso Physical Design Pittsburgh and San Jose teams, for their willingness to try something new. We would also like to thank our next round of volunteer teams: the Virtuoso ViVA team, Interconnect MIS team, Physical Design and Interconnect Sophia teams, Schematics and Integ Noida team, and the ICFT Connectivity team. And of course, thanks to our partners at the SEI at Carnegie Mellon who have worked with us in training and deployment: Jim McHale, Jim Over, and Greg Such.

5.8 Author Biographies

Elias Fallon
Engineering Director
Cadence Design Systems, Custom IC and Simulation R&D

Elias Fallon is an engineering director in the Custom IC and Simulation R&D group at Cadence Design Systems, Inc. He has worked at Cadence since 2004 when Cadence acquired Neolinear, Inc. Fallon held a variety of roles at Neolinear since 1997. Fallon holds a BS and MS in Electrical and Computer Engineering from Carnegie Mellon University.

Lee Gazlay
Engineering Group Director
Cadence Design Systems, Custom IC and Simulation R&D

Lee Gazlay is an engineering group director in the Custom IC and Simulation R&D group at Cadence Design Systems, Inc. He has more than 25 years of experience in the electronic design automation industry.

5.9 References

[Ben-Yaacov 2001]
Ben-Yaacov, G. & Gazlay, L. "Real World Software Testing at a Silicon Valley High-Tech Software Company" STAREAST 2001 Software Testing Conference, Orlando, Florida, May 14-18, 2001.

[Burton 2006]
Burton, D. & Humphrey, W. S. 2006. "Mining PSP Data." *Proceedings of the TSP Symposium.* New Orleans, LA. September 2006.
http://www.sei.cmu.edu/tspsymposium/2009/2006/mining.pdf

[Humphrey 2009]
Humphrey, W. *The Watts New? Collection, Columns by the SEI's Watts Humphrey* (CMU/SEI-2009-SR-024), "Learning from Hardware: Design and Quality" pp. 99-103. Software Engineering Institute, Carnegie Mellon University, 2009.
http://www.sei.cmu.edu/library/abstracts/reports/09sr024.cfm

[Wikipedia 2012]
Wikipedia, The Free Encyclopedia, "Moore's law," July 2012.
http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=501537223.

6 A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report

Shigeru Kusakabe, Kyushu University
Yoichi Omori, Kyushu University
Keijiro Araki, Kyushu University

6.1 Abstract

Software process is important for producing high-quality software and for its effective and efficient development. The Personal Software Process (PSP) provides a method for learning a concept of personal software process and for realizing an effective and efficient process by measuring, controlling, managing, and improving the way we develop software. PSP also serves as a vehicle to integrate advanced software engineering techniques, including formal methods, into one's own software development process. While formal methods are useful in reducing defects injected into a system, by mathematically describing and reasoning about the system, engineers may have difficulties integrating formal methods into their own software development processes. We propose an approach in which engineers use PSP to introduce formal methods into their software processes. As our initial trial, we followed one of our graduate students as he tried to improve his personal process with this approach. He measured and analyzed his own process data from PSP for Engineers-I, and proposed and experimented with an improved software process with a formal method, the Vienna Development Method (VDM). The experimental results indicate he could effectively reduce defects by using VDM.

6.2 Introduction

As software process is important for producing high-quality software and for its effective and efficient development. Improved software process leads to improved product quality through more effective and efficient development. The PSP provides a method for learning a concept of personal software process and making it more effective and efficient by using measurement and analysis tools for understanding our own skills and improving personal performance [Humphrey 1996, 2005].

Inspired by the introduction of process education in Kyushu Institute of Technology (KIT) in cooperation with the Carnegie Mellon Software Engineering Institute (SEI) [Katamine 2011], in Kyushu University, we started process education and research from a laboratory level activity [Kozai 2009]. Since then, we have been expanding process education activity. We currently have two PSP instructors, thanks to the support of the process education group of KIT, and we offer PSP for Engineers I & II in the curriculum of our graduate school. In addition to introducing a process concept, we also use PSP in attempts to leverage our research and education activity.

PSP is useful as a learning vehicle for introducing process concepts, and further as a base of software process improvement to integrate advanced software engineering techniques, including formal methods, into software development processes. Formal methods are mathematically based techniques useful in reducing defects injected into computer-based systems by mathematically describing and reasoning about the systems. We have various kinds of formal methods such as

model checking and model-oriented formal specification. One of the challenges of using formal methods, in addition to selecting a specific formal method to use, is determining when and how to use the formal method in actual software development.

We propose an approach in which developers use PSP as a framework of software process improvement to introduce formal methods into their software process for realizing effective and efficient development of high-quality software. Since the main focus is defect prevention and removal in the Design phase, the defect log in PSP is useful in analyzing details of defects from important points of view such as their type and cost. As our initial trial, we followed one of our graduate students as he tried to improve his personal process with this approach. He measured and analyzed his own process data using the course materials of PSP for Engineers I, and proposed an improved process with a formal method, Vienna Development Method (), a model-oriented formal method. He used the improved process for a few exercises in the course material of PSP for Engineers II, and experimental results indicate he could effectively reduce defects by using VDM.

6.3 Personal Software Process

PSP provides an improvement framework that helps us to control, manage, and improve the way we work [Humphrey 2005]. The phases in PSP that we complete in building software are Plan, Detailed Design, Detailed Design Review, Code, Code Review, Compile, Unit Test, and Post Mortem. Each phase has a set of associated scripts, forms, and templates, some of which are used over multiple phases. For each phase, the engineer collects data on the time spent and the defects injected and removed. Size is collected in the Post Mortem phase. Other data are calculated, including size and time estimating error, cost-performance index, defects injected and removed per hour, personal yield, appraisal and failure cost of quality, and the appraisal to failure ratio. The defect data consists of the defect type, the time to find and fix the defect, the phase in which the defect was injected, the phase in which it was removed, and a brief description of it.

In PSP, we see how the use of forms in a well-defined process help us to gather and use process data. We can use PSP data to consider how to define and improve our own processes. For example, the PIP, or Process Improvement Proposal form, lets us record problems, issues, and ideas to use later in improving our processes. To ensure quality, we construct, use, and improve checklists for design and code review by analyzing defect data from the PSP exercises. PSP 2.1 introduces design specification and analysis techniques. PSP provides four design templates to increase confidence in design: internal-static, internal-dynamic, external-static, and external-dynamic. We learn the importance of focusing on quality and how to efficiently review our programs. From our own data, we see how checklists can help us to effectively review design and code as well as develop and modify these checklists as our personal skills and practices evolve.

In their paper on design defect analysis, Vallespir and Nichols focused on the analysis of defects injected during the Design and Code phase, checking properties such as defect types and individual defect times [Vallespir 2011]. Their analysis shows not only that the defects injected during design are the most expensive to remove in Test but also that these are easy to remove in the Review phases. The difference of cost between review and test was remarkable in that analysis: it cost five times more to remove a defect in test than it did to remove that same defect during review.

PSP is useful as a learning vehicle for the basic concept of software process and its improvement, and further as a base of advanced software process improvement in which we integrate various software engineering techniques, including formal methods, into a software development process.

6.4 Formal Methods

Formal methods are mathematically based techniques useful in reducing defects injected into computer-based systems by mathematically describing and reasoning about the systems. We have various kinds of formal methods such as model checking and model-oriented formal specification. Although they are typically useful at the earlier stages of design and complement fault removal techniques like testing, some of them are specifically effective for limited aspects and others are generally effective for a wide range of aspects. A growing number of software engineers seem to have interests in formal methods. This is partly because a wider range of software supports important social infrastructure in which software problems may heavily impact our society. Formal methods are helpful in increasing confidence in software, and international standards such as ISO/IEC 15408 and IEC61508 mention formal methods.

Formal methods are effective in increasing our confidence in specifications and designs, and in decreasing defects in software development. However, formal methods are not widely used in actual software development as might be expected. There are still problems in introducing formal methods to a majority of practitioners, although successful instances of introduction of formal methods have been reported already [Kurita 2008]. In order to promote widespread introduction of formal methods, for example in Japan, several organizations such as SEC (Software Engineering Center) of IPA (Information-technology Promotion Agency) and DSF (Dependable Software Forum) provide supporting materials and seminars. One of the major problems seems to be that many managers and developers have no idea what kind of formal methods are useful for them. While we can use formal methods for the specification, development, and verification of the target systems, we have various choices of formal methods and the ways to use them. Problems also include integration into current development practice and education as well as quantitative issues such as cost effectiveness. Even if they select a specific method, they may have no idea when and where to use the formal method, and whether it is cost effective or not.

Our ideal guideline to introduce formal methods into software development processes is a customizable one, based on a well-defined, measured software process. By using such process, we can effectively and efficiently employ formal methods at the specific points where we have concerns on quality by considering cost-performance tradeoff.

Formal methods can be used at different levels of formality:

- Level 0: In this lightweight formal methods level, we develop a formal specification and then a program from the specification informally. This may be the most cost-effective approach in many cases.
- Level 1: We may adopt formal development and formal verification in a more formal manner to produce software. For example, proofs of properties or refinement from the specification to an implementation may be conducted. This may be most appropriate in high-integrity systems involving safety or security.

- Level 2: Theorem provers may be used to perform fully formal machine-checked proofs. This kind of activity may be very expensive and is only practically worthwhile if the cost of defects is extremely expensive.

Lightweight formal methods are most cost effective and likely to be adopted in a wider range of actual development. In lightweight formal methods, we do not rely on very rigorous means, such as theorem proofs. Instead, we use adequate, less rigorous means, such as evaluation of pre/post conditions and testing specifications, to increase confidence in specifications, while the specific level of rigor depends on the goal of the project.

One instance of lightweight formal methods is VDM which uses model oriented formal specification languages, such as VDM-SL and VDM++, to write, verify, and validate specifications of the products. VDM languages have a tool, VDMTools, with functionalities of syntax check, type check, interpreter, and generation of proof-obligations. In VDM, we can write explicit-style executable specifications as well as implicit-style non-executable specifications. For explicit-style executable specifications, we can use the interpreter to evaluate pre/post conditions in the specifications and test the specifications. Since we can write specifications at different levels of abstraction, including detailed level close to implementation level, we can use VDM languages in a way close to popular development styles. One of the features of VDM is that it enables us to write and execute specifications like programs in programming languages.

We propose an approach in which developers use PSP as a framework of software process improvement to introduce formal methods into their software process for realizing effective and efficient development of high-quality software. Since the main focus is defect prevention and removal in the Design phase, the defect log in PSP is useful in analyzing details of defects such as their type and cost.

6.5 Process Improvement with Formal Methods

The important goals of software process improvement include making software development more effective and efficient as well as improving software quality. In this study, we mainly customize the Design phase to achieve fewer defects and enhanced productivity by using formal methods, as formal methods are typically useful at the earlier stages of design and complement fault removal techniques such as testing.

By measuring the time that tasks take and the number of defects they inject and remove in each process phase, engineers learn to evaluate and improve their personal performance. Engineers should understand the defects they inject before they attempt to improve their process from the viewpoint of quality. As engineers learn to track and analyze defects in the PSP exercises, they gather data on the phases when the defects were injected and removed, the defect types, the fix time, and defect descriptions.

Defects may come from simple oversights, misunderstandings, and human errors, as well as complicated logic. Many of these defects are caused by an inadequate convention to represent design. Poor notation frameworks in design can make designers write vague or ambiguous design representations which prevent designers from examining the design representation consistently or in great enough detail, lead developers to design in the coding phase, and thus become a significant source of errors.

Although PSP provides four design templates to increase confidence in design, we try to use formal methods as formal specification languages, and their supporting tools are effective in reducing those defects discussed above. In this study, we use VDM, which has formal specification languages and a tool, VDMTools, for syntax check, type check, interpreter, and generation of proof-obligations. In VDM, we can write implicit specifications focusing on pre-conditions and post-conditions without explicitly describing implementation logic, as well as explicit specifications that include explicit descriptions of implement logic. We expect implicit specifications in VDM to correspond to functional specification templates in PSP, and explicit specifications in VDM to correspond to logic specification templates in PSP.

6.6 Case Report

In our initial trial, we followed one of our graduate students as he tried to improve his personal process with our approach: process improvement using a formal method based on PSP process extension framework. He used the course materials from PSP for Engineers I, and measured and analyzed his own process data from the PSP course. The focus was on defects frequently injected and expensive to fix. He proposed an improved software process with VDM to facilitate defect prevention and removal in the Design phase. He used his new process for a few exercises in the course material of PSP for Engineers II. The experimental results indicate he could effectively reduce defects by using VDM. We explain more details in the rest of this section.

In this trial, the graduate student developed four programs in PSP for Engineers I. He used process data from these programs as the baseline for improvement. He analyzed the process data of the baseline process based on the defect types defined in PSP, and the cause of the defects injected during the Design phase of the PSP programs. The goal of the analysis was to understand where defects and their causes were injected, the types of defects injected in the Design phase, when those defects were removed, and the effort required to find and fix those defects. According to the results of the analysis, he decided to focus on the defect types that were most frequently injected and expensive to fix. Table 15 shows the defect types and explanatory description with average fix times in minutes.

	Defect Type	Description	Average Fix Time (Min.)
I-1	Interface	Insufficient Refinement	15.8
F-1	Function	Looping Control	10.3
F-2	Function	Logic	6.8

Table 15: Defect Types Frequently Injected and Expensive to Fix

In the baseline process, the graduate student used UML in the Design phase. However, he felt diagrams in UML were difficult to check rigorously. Then he decided to use VDM++ in the detailed Design phase and developed two programs. In introducing VDM++ into PSP, we can employ a lightweight approach, which corresponds to the so-called level-0 approach, since formal methods can be used at different levels. Level 0 is most cost effective. In VDM, we use formal specification languages at different abstraction levels. We can develop high-level designs and also detailed designs in a lightweight way without rigorous proof. We can use the tool VDMTools for syntax and type check of the specification. In addition, we can execute our specifications if they are written as explicit executable ones.

To prevent injection of defects of the types described above, the student extended the baseline process to include the following steps.

1. Write signature of methods in VDM++ in detailed Design phase and use VDMTools for syntax and type check to prevent injection of defects of I-1 type.
2. Describe sequence handling control in VDM++ to prevent injection of defects of F-1 type.
3. Write explicit VDM++ specifications for selected parts and using animation of VDMTools.

For exercise programs in PSP2, the student made a program design prior to coding and described the design using VDM++ instead of the default templates: functional, logical, operational, and state templates. He then performed a checklist-based personal review and tool-based check of the design to identify and remove design defects before beginning to write code. Figure 32 shows the comparison of time ratio spent in each phase. As we can see from the figure, the developer spent more in design and design review and less in coding and testing.

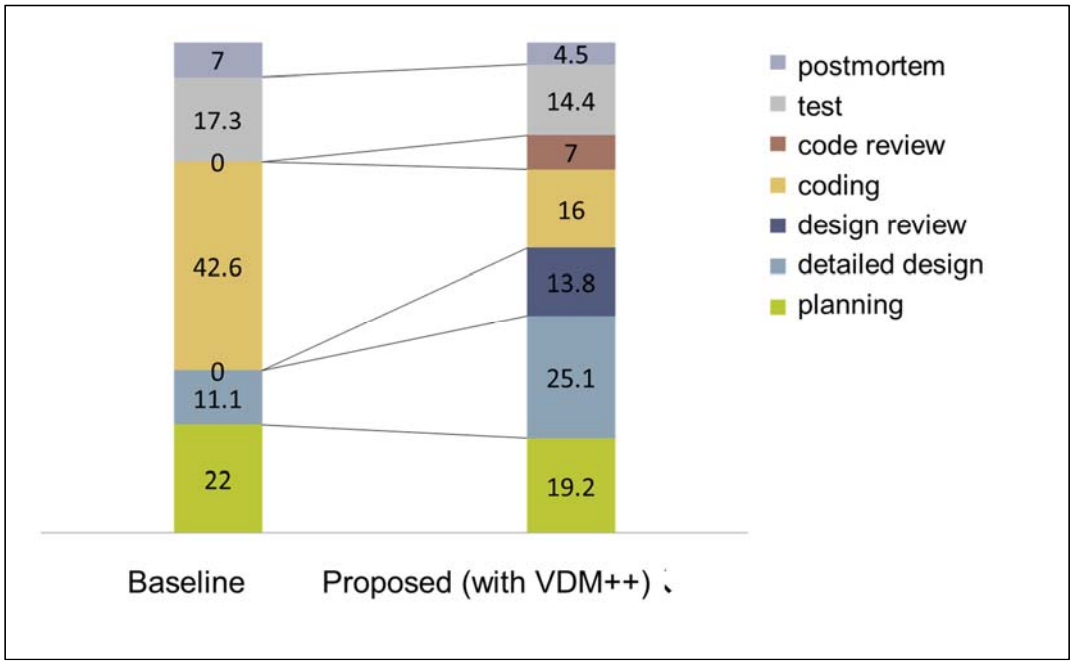


Figure 32: The Comparison of Time Ratio Spent in Each Phase

Figure 33 shows the comparison of defect density, as number of defects per one thousand lines of code, between the baseline process and the extended process with VDM++.

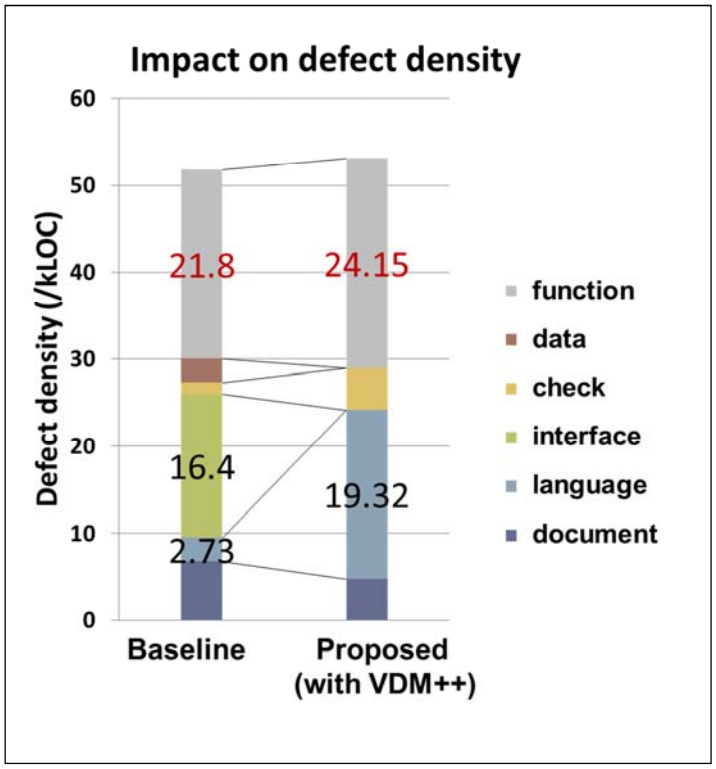


Figure 33: The Comparison of Defect Density, Number of Defects Per One Thousand Lines of Code, Between the Baseline Process and the Extended Process with VDM++

As we can see from the figure, rigorously described interfaces in formal languages such as VDM++ are effective in reducing defects, while there seems to be no difference in total defect density between the baseline and the extended process. The defects of language type were injected mainly because the student was not as familiar with the programming language used in this trial. We can expect these kinds of defects to be reduced if the developer is familiar with the programming language. Please note that we could reduce defects of interface type, one of the focused defect types.

We conclude that rigorously described interfaces in formal languages such as VDM++ are effective in reducing defects. We could reduce defects of the type we focused on without causing a decrease in productivity.

6.7 Concluding Remarks

We reported our initial trial of the introduction of formal methods into personal process based on PSP. According to the process data in the trial, the developer spent more time in Design and less time in Test. He successfully reduced the number of defects he had focused on without decreasing his productivity.[3] He had the impression that without a disciplined process like PSP he could not have made a process improvement plan with formal methods. We will continue to work on this

[3] Note that proficiency in programming language and software development skills might affect the results.

kind of effort, as this is a report of our initial trial and the coverage of the issues and the experiment was limited. We also have a plan to extend our approach to TSPi as our future work.

6.8 Author Biographies

Shigeru Kusakabe
Associate Professor
Graduate School of Information Science and Electrical Engineering, Kyushu University

Shigeru Kusakabe received his B.E., M.E., and D.E. degrees from Kyushu University in 1989, 1991, and 1998 respectively. He was a research assistant at Department of Information Systems, Kyushu University from 1991 to 1998. He has been an associate professor since 1998 in the Department of Computer Science and Communication Engineering. His research interests include functional languages, multithreading, operating systems, cloud computing, applied behavior analysis, and formal methods in addition to software development process. He is a member of ACM, IEEE-CE, IEICE, IPSJ, SEA, and J-ABA. He has been a PSP instructor since 2010.

Yoichi Omori
Assistant Professor
Graduate School of Information Science and Electrical Engineering, Kyushu University

Yoichi Omori is an assistant professor of the graduate school and faculty of information science and electrical engineering at Kyushu University. He received a PhD in engineering from Nara Institute of Science and Technology. His research interests include requirement engineering, formal modeling language, dynamic model verification, and their developing environments. Dr. Omori is building a requirement managing tool which supports translation from requirements in natural languages into a formal language. He also teaches courses on project-based learning with corporations in the graduate school.

Keijiro Araki
Professor & Vice Dean
Graduate School of Information Science and Electrical Engineering, Kyushu University

Keijiro Araki received MS. and PhD degrees in Computer Science and Communication Engineering from Kyushu University. His research interests include formal approaches to software development, formal specification, and dependable systems. He is a member of the Science Council of Japan, chair of the Kyushu Chapter of Information Processing Society of Japan, and chair of the Kyushu Chapter of the Society of Project Management.

6.9 References

[Humphrey 2005]
Humphrey, Watts S. *PSP: A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005. http://www.sei.cmu.edu/library/abstracts/books/0321305493.cfm

[Humphrey 1996]
Humphrey, Watts S. "Using A Defined and Measured Personal Software Process." *IEEE Software 13,* 3 (1996): pp.77–88.

[Katamine 2011]
Katamine, Keiichi; Umeda, Masanobu; Hashimoto, Masaaki; & Akiyama, Yoshihiro. “Changing Software Management Culture from Academic,” 12-18. *TSP Symposium 2011*. Atlanta GA, September 2011. Software Engineering Institute, 2011.

[Kozai 2009]
Kozai, Ken; Kusakabe, Shigeru; Omori, Yoichi; & Araki, Keijiro. *Introducing formal methods into measurable personal software development processes* (Vol. 2009-SE-163-21). *IPSJ SIG*, 2009.

[Kurita 2008]
Kurita, T.; Chiba, M.; & Nakatsugawa, Y. “Application of a formal specification language in the development of the mobile felica IC chip firmware for embedding in mobile phone.” *FM 2008: Formal Methods* (2008): 425-429.

[Vallespir 2011]
Vallespir, Diego & Nichols, William. “Analysis of Design Defects Injection and Removal in PSP,” 19–25. *Proceedings of the TSP Symposium 2011: A dedication to excellence*. Atlanta, GA, September 2011. Software Engineering Institute, Carnegie Mellon University, 2011.

7 A Cross Course Analysis of Product Quality Improvement with PSP

Fernanda Grazioli, Universidad de la República
William Nichols

7.1 Introduction and Background

These days, more and more businesses develop, combine, and include software in their products in different ways. Companies need to develop software to support the design, manufacture, or delivery of the products and services they provide. Therefore, although they might not realize it, all businesses are becoming software businesses. As the software component of their business grows, schedule delays, cost overruns, and quality problems caused by software become their main business problems. This is why, despite their best management efforts, companies find their risk of failure increasing along with the increase in the size or complexity of the software they produce.

Software products are made of hundreds to millions of lines of code, each one handcrafted by a software engineer. Software businesses depend on people, so their technical practices and experience strongly influence the outcome of the development process.

The Personal Software Process (PSP) is a defined and measured software process designed to be used by an individual software engineer. The PSP directly addresses software business needs by improving the technical practices and individual abilities of software engineers, and by providing a quantitative basis for managing the development process. By improving individual performance, PSP can improve the performance of the organization.

For many years, the Software Engineering Institute (SEI) has trained software engineers in PSP. Over that period, the course format has changed twice. Several versions of the course use the same exercises, but introduce process steps in modified sequences. An earlier version of the course has several published studies demonstrating improvement in developer performance with process insertion, but the retrospective analysis left some threats to external validity [Paulk 2006; Hayes 1997; Wohlin 1998; Rombach 2008; Kemerer 2009; Paulk 2010]. One threat is the confounding of process insertion with the gaining of domain experience as related programs are developed. A related threat is that observations might alter the subject performance as in the Hawthorne effect [Mayo 1949]. Moreover, there are not yet studies about how the latest two course versions are working, nor are there studies about how different approaches to introducing process correlate with performance and quality results in PSP courses. The PSP community and the SEI need to know how effectively these courses work. The academic and industrial communities need assurance that the process can be taught effectively and that the process insertion would have positive and substantial benefits.

Given this situation, the objective of this study is to use the PSP data from the latest two course formats to determine whether reviews and design improve product quality, or if such improvement is only a consequence of gaining experience in the problem domain. We measure quality as the quantity of defects found per KLOC in Unit Testing. Defect counts and measures of

defect density (i.e., defects per KLOC) have traditionally served as software quality measures. The PSP uses this method of measuring product quality as well as several process quality metrics. The consequence of high defect density in software engineering is typically seen in the form of bug-fixing or rework effort incurred on projects. Typical defect densities of delivered products range from one to five defects/KLOC [Davis 2003].

7.1.1 Concept Introduction on PSP Courses

The PSP courses incorporate what has been called a "self-convincing" learning strategy that uses data from the engineer's own performance to improve learning and motivate use. The last two course versions introduce the PSP practices in steps corresponding to six PSP process levels. The older version name is "PSP for Engineers I/II (PSPI/II)" and the latest version name is "PSP Fundamentals and Advanced (PSP Fund/Adv)."

Each level builds on the developed capabilities and historical data gathered in the previous level. Engineers learn to use the PSP by writing seven or eight programs (depending on the course version), and by preparing written reports. Engineers may use any design method or programming language in which they are fluent. The programs typically contain around one hundred lines of code (LOC) and require a few hours on average to be completed. While writing the programs, engineers gather process data that are summarized and analyzed during a postmortem phase. There are three basic measures in the PSP: development time, defects, and size. All other PSP measures are derived from these three basic measures.

During the course, the students were given eight or seven exercises (eight in PSPI/II and seven in PSP Fund/Adv), which were mainly programs for statistical calculations. PSP has a maturity framework that shows its progression on improvement phases, also called levels. Students completed 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
- PSP1.1: Task planning and schedule planning
- PSP2: Techniques to review code and design
- PSP2.1: Introduction of design templates

Table 16 shows which PSP level is applied on each program assignment, for each course version.

Table 16: PSP Levels for each Program Assignment

Program Assignment	PSP Fund/Adv	PSP I/II
1	PSP 0	PSP 0
2	PSP 1	PSP 0.1
3	PSP 2	PSP 1
4	PSP 2	PSP 1.1
5	PSP 2.1	PSP 2

Program Assignment	PSP Fund/Adv	PSP I/II
6	PSP 2.1	PSP 2.1
7	PSP 2.1	PSP 2.1
8	-----	PSP 2.1

7.2 Data Set and Statistical Model

We used data from the eight-program course version, PSPI/II, taught between June 2006 and June 2010. Additionally, we used data from the seven-program course version, of PSP Fund/Adv, taught between December 2007 and September 2010. These courses were taught by the SEI at Carnegie Mellon University or by SEI partners, by a number of different instructors in multiple countries.

We began with 347 subjects in total, 169 from the PSP Fund/Adv course and 178 from the PSPI/II course. From this we made several cuts and ran data-cleaning algorithms to include only the students who had completed all programming exercises, in order to clean and remove errors and questionable data.

To determine the cuts on the data set, we first developed an integrated data storage model. We designed that model to support the analysis and the assessment of data quality, based on the data quality theory. Data quality is an investigation area that has generated a great workload in the last years and it is mainly focused on defining the aspects of data quality [Batini 2006; Lee 2002; Neely 2005; Strong 1997] and on proposing techniques, methods and methodologies to the measurement and treatment of the data quality [Batini 2006; Lee 2002;Wang 1995].

The first step toward identifying quality problems was to understand the reality and context to be analyzed. This includes the Personal Software Process in itself [Humphrey 1995], exploring the tool for recording data and the model of the database, in addition to the Grading Checklists used by instructors for the correction of exercises performed during the course. Afterwards, we analyzed the dimensions and quality factors proposed by Batini and Scannapieco [Batini 2006] to this set of data, which are interesting to measure and consider. In this way, we thoroughly identified and defined possible quality problems that the data under study might contain, we implemented the algorithms required for cleaning and collecting the metadata, and finally, we executed those algorithms. Major data quality problems were related to the consistency, accuracy, completeness, and uniqueness dimensions. This meant that following that data quality process, our data set was reduced to 93 subjects in total, 45 from the PSP Fund/Adv course and 47 from the PSPI/II course.

Differences in performance between engineers are typically the greatest source of variability in software engineering research, and this study is no exception. However, the design of the PSP training class and the standardization of each engineer's measurement practice allow the use of statistical models that are well suited for dealing with the variation among engineers.

In the summarized analyses presented, we studied the changes in engineers' data over seven programming assignments. Rather than analyzing changes in group averages, this study focuses on the average changes of individual engineers. Some engineers performed better than others from the first assignment, and some improved faster than others during the training course. To

discover the pattern of improvement in the presence of these natural differences between engineers, the statistical method known as the repeated measures analysis of variance (ANOVA) is used [Tabachnick 1989]. In brief, the repeated measures analysis of variance takes advantage of situations where the same people are measured over a succession of trials. By treating previous trials as baselines, the differences in measures across trials (rather than the measures themselves) are analyzed to uncover trends across the data. This allows for differences among baselines to be factored out of the analysis. In addition, the different rates of improvement between people can be viewed more clearly. If the majority of people change substantially (relative to their own baselines), the statistical test will reveal this pattern. If only a few people improve in performance, the statistical test is not likely to suggest a statistically significant difference, no matter how large the improvement of these few people.

7.3 Analysis and Results

First, we define the following variables and terms:

- Subject –student who performs a complete PSP course.
- Course Type –a PSP course version. It can be PSP Fund/Adv or PSPI/II.
- Program Assignment or Program Number –an exercise that a student has performed during the PSP course. It can be 1, 2, 3, 4, 5, 6, 7 or 8.
- PSP Level –one of the six process levels used to introduce the PSP in these course versions. It can be PSP0, PSP0.1, PSP1, PSP1.1, PSP2, or PSP2.1. Each program assignment has a corresponding PSP level according to the PSP course version. Since we wanted to analyze the introduction of concepts during the courses, we group PSP0 and PSP0.1, we group PSP1.0 and PSP1.1, and we analyze PSP2.0 and PSP2.1 separately. So the PSP Level variable can be seen in plots as 0, 1, 2, or 2.1, respectively.
- Defect Density in Unit Testing (DDUT) – 1000 · Total defects removed in testing / Actual added and modified LOC

As we stated in the introduction, the objective of this study is to use the PSP data from the latest two course formats to demonstrate whether reviews and design improve the quality of a product in test. And we use defect density as a measure of quality, so we define defect density in unit testing as the independent variable in our analyses. Figure 34 shows a bar-whisker chart of DDUT grouped by course type and PSP level. Figure 35 shows a bar-whisker chart of DDUT, but in this case grouped by course type and program assignment. These charts are descriptive and allow us to get a clearer idea of the defect density behavior.

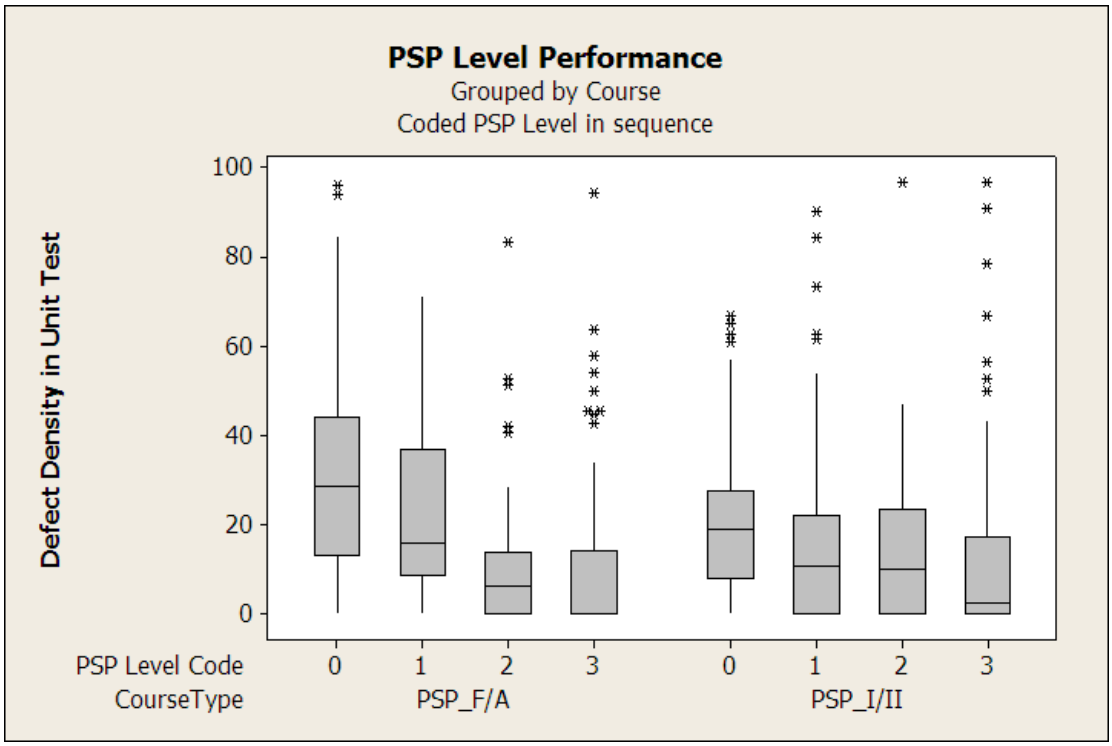


Figure 34: Defect Density in Unit Testing grouped by Course Type and PSP Level

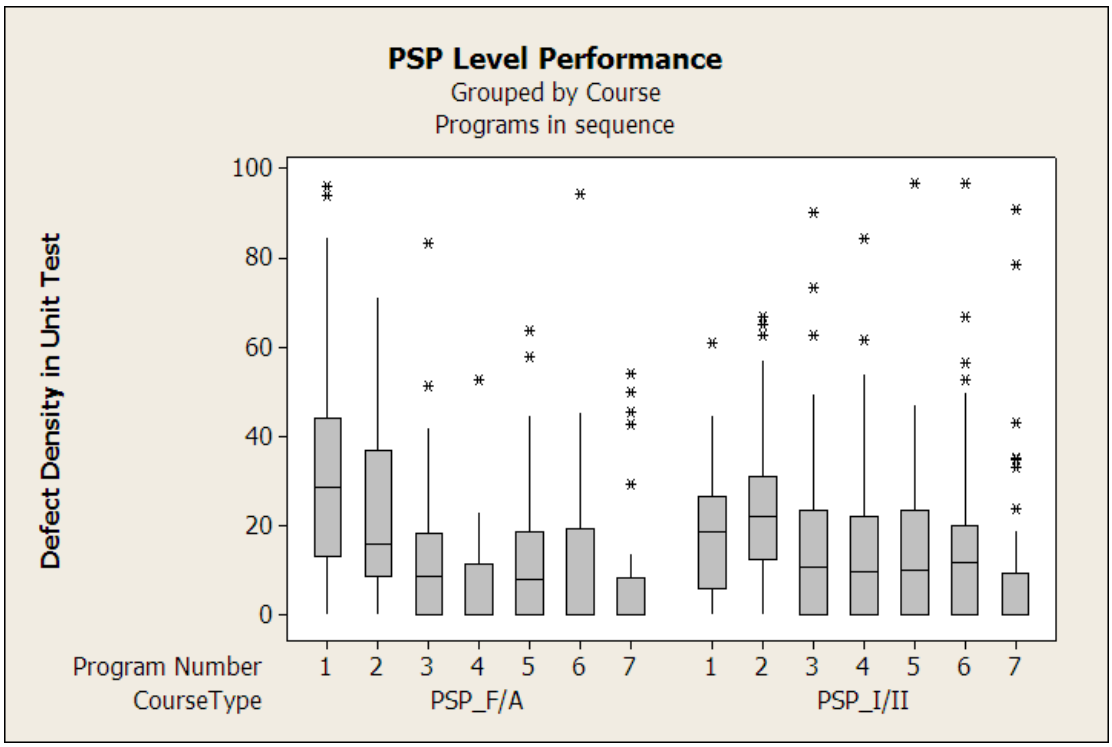


Figure 35: Defect Density in Unit Testing Grouped by Course Type and Program Assignment

To reach that objective, we considered a direct way through an ANCOVA analysis, using PSPI/II and PSP Fund/Adv as categorical values to segment the data into two parts: the program assignment as the independent variable and the PSP level as the hidden variable. But we could not

come to a conclusion because the correlation between the PSP level and the program assignment was strong. Also, using ordinal program numbers in ANCOVA may not be a sound approach. Correlations between the factors are a strong anti-indication for ANCOVA [Tabachnick 1989]. Therefore, we decided to create a more indirect procedure and analyze the results based on specific differences between the two courses using the PSP level.

We next developed an indirect procedure to examine relationships between program number, PSP level, and performance in the data. It consists of three steps, each one based on an ANOVA analysis using defect density in unit testing as the independent variable.

Before running any ANOVA, we checked to see if the data satisfy all assumptions for the correct use. The only assumption that could not be fully satisfied was the normality assumption. We transformed the data into normal form using different techniques as suggested in [Tabachnick 1989], and the one that worked better was a log transformation and adding a constant to the zero observations. Nonetheless, some deviations from normality persisted at very low defect levels. Recent works recommend not to log-transform count data [O'Hara 2010] for two reasons. First, low counts cause distortions. Second, such transformation typically does not affect the results if the data is unimodal. The data satisfies the unimodal mounded condition, and the fact that transformed and untransformed PSP data provides comparable results has been previously observed with [Hayes 1997]. We did our analysis considering both transformed and untransformed data.

The first step of the analysis procedure consisted of performing a series of one-way ANOVA between each program number using course as the grouping factor. This establishes whether the assignments have different DDUT means between the courses. If there are not different means between courses, then the analysis is done because the program-by-program results for the two courses do not differ. We ran seven tests, one for each program number and, since we found significant differences, we proceeded to the next step.

The second step of the analysis procedure consisted of performing two-way (repeated) ANOVA. Separate, repeated ANOVA analyses grouping by program number were performed for each course type. We applied the two-tailed significance test at 0.05, which is equivalent to a 0.025 significance level for a one-tailed test. Table 17 and Table 18 summarize the ANOVA discussed above for this step for the courses PSP Fund/Adv and PSP I/II, respectively. A third two-way ANOVA grouping by course and program number was performed on the combined data, and the results are summarized in Table 19.

Both the separate course data and the combined data showed a general downward trend in defect level with program number, irrespective of process level. This was as we expected based on the correlation between PSP level and program number. We found no statistical significance between consecutive programs for either the PSP Fund/Adv or the PSP I/II. However, for the combined data set, we found significant reduction in defect levels between programs 2 and 3, and 3, and 4. The significance in the combined set results from the increased sample size.

In the PSP Fund/Adv course we found that there is significance between Program 1 and Programs 3, 4, 5, 6, and 7. We interpret that this shows improvements between PSP0 and PSP2 or PSP2.1 (depending on which program from 3 to 7). In regard to PSP I/II, we found that there is

significance between Program 1 and Program 7, and this is consistent with improvements between PSP0 and PSP2.1.

We also found that there is significance in PSP Fund/Adv between Program 2 and Program 3, 4, 5, 6, and 7. This makes sense because it shows improvement between PSP1 and PSP2 or PSP2.1 (depending on which program from 3 to 7). In regard to PSP I/II, we found that there is significance between Program 2 and Program 4, 5, 6, 7. This also makes sense because it shows improvement between PSP0 and PSP1 or PSP2 or PSP2.1 (also depending on which program from 4 to 7).

Table 17: ANOVA Outputs for Program Assignment Comparison in PSP Fund/Adv

PSP Fund/Adv				
Program Assignment (I)	**Program Assignment (J)**	**PSP Level**	**Mean difference (I-J)**	**Sig.**
1	2	1	*,154*	*,999*
	3	2	*1,364*	*,003*
	4	2	*2,348*	*,000*
	5	2.1	*1,602*	*,000*
	6	2.1	*2,139*	*,000*
	7	2.1	*2,347*	*,000*
2	3	2	*1,210*	*,012*
	4	2	*2,193*	*,000*
	5	2.1	*1,448*	*,001*
	6	2.1	*1,985*	*,000*
	7	2.1	*2,192*	*,000*
3	4	2	*,983*	*,082*
	5	2.1	*,237*	*,994*
	6	2.1	*,774*	*,301*
	7	2.1	*,982*	*,082*
4	5	2.1	*-,745*	*,347*
	6	2.1	*-,208*	*,997*
	7	2.1	*-,001*	*1,000*
5	6	2.1	*,537*	*,731*
	7	2.1	*,744*	*,349*
6	7	2.1	*,207*	*,997*

Table 18: ANOVA Outputs for Program Assignment Comparison in PSP I/II

PSP I/II				
Program Assignment (I)	**Program Assignment (J)**	**PSP Level**	**Mean difference (I-J)**	**Sig.**
1	2	0.1	*-,485*	*,820*
	3	1	*,502*	*,795*
	4	1.1	*,730*	*,381*
	5	2	*,816*	*,248*
	6	2.1	*,948*	*,109*
	7	2.1	*1,517*	*,001*
2	3	1	*,987*	*,083*
	4	1.1	*1,216*	*,012*
	5	2	*1,301*	*,005*
	6	2.1	*1,434*	*,001*

PSP I/II				
Program Assignment (I)	**Program Assignment (J)**	**PSP Level**	**Mean difference (I-J)**	**Sig.**
	7	2.1	*2,003*	*,000*
3	4	1.1	*,228*	*,995*
	5	2	*,314*	*,975*
	6	2.1	*,446*	*,871*
	7	2.1	*1,015*	*,067*
4	5	2	*,0854*	*1,000*
	6	2.1	*,2179*	*,996*
	7	2.1	*,786*	*,290*
5	6	2.1	*,132*	*1,000*
	7	2.1	*,701*	*,433*
6	7	2.1	*,569*	*,682*

Table 19 shows a summary of the ANOVA for the combined data.

Table 19: ANOVA Outputs for Program Assignment Comparison Combined Course Data

Program Assignment (I)	**Program Assignment (J)**	**Mean difference (I-J)**	**Sig.**
1	2	*-,166*	*,509*
	3	*,933*	*,000*
	4	*1,539*	*,000*
	5	*1,209*	*,000*
	6	*1,544*	*,000*
	7	*1,932*	*,000*
2	3	*1,099*	*,000*
	4	*1,705*	*,000*
	5	*1,375*	*,000*
	6	*1,710*	*,000*
	7	*2,098*	*,000*
3	4	*,606*	*,016*
	5	*,276*	*,271*
	6	*,611*	*,015*
	7	*,999*	*,000*
4	5	*-,330*	*,188*
	6	*,005*	*,985*
	7	*,393*	*,117*
5	6	*,335*	*,182*
	7	*,723*	*,004*
6	7	*,388*	*,122*

As a summary of this step, we can say that for each course we found significant difference only between assignments with different PSP levels. For the combined course we found a significant difference between programs 2 and 3 (introduced in PSP level 2 in PSP Fundamentals) and programs 3 and 4 (PSP the second use of level 2 in Fundamentals and PSP level 1.0 to 1.1 estimation by parts, in PSP I).

According to the design and review techniques introduced in the corresponding PSP levels, we expected these improvements. Figure 36 shows the estimated marginal means of the log-

transformation of defect density in unit testing versus program number, for both courses. The graphic shows how the two courses perform differently. The declining defect level is more consistent and larger in PSP Fundamentals through the introduction of PSP 2.0. Defect levels appear to be more consistent by the end of the courses.

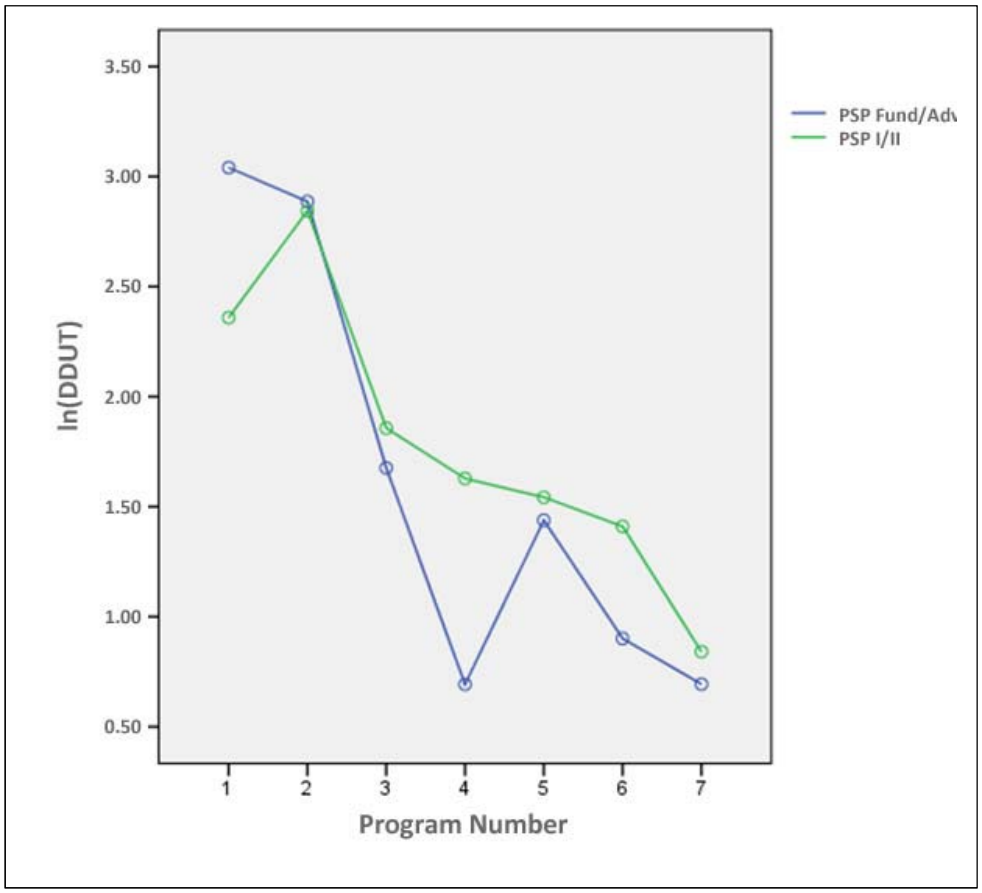


Figure 36: Comparison of Estimated Marginal Means of Ln(DDUT) versus Program Number between PSP Fund/Adv and PSP I/II

The third and last step of the analysis procedure consists of performing a two-way ANOVA grouping by PSP level and course to set bounds on the importance of PSP level as a predictor.

After this test, we found that there is significant difference between PSP0 and PSP1, between PSP0 and PSP2, and also between PSP0 and PSP2.1. Looking at it in more detail, results show that

- PSP1 was a factor of 0.2 more effective than PSP0 at an alpha level of 0.05 with a confidence range of the differences of [0.074, 0.939].
- PSP2 was a factor of 0.4 more effective than PSP0 at an alpha level of 0.05 with a confidence range of the differences of [1.028, 1.888].
- PSP2.1 was a factor of 0.45 more effective than PSP0 at an alpha level of 0.05 with a confidence range of the differences of [1.373, 2.133].

We also found that there is significant difference between PSP1 and PSP2, and between PSP1 and PSP2.1. Looking at it in more detail, results show that

- PSP2 was a factor of 0.22 more effective than PSP1 at an alpha level of 0.05 with a confidence range of the differences of [0.521, 1.381].
- PSP2.1 was a factor of 0.28 more effective than PSP1 at an alpha level of 0.05 with a confidence range of the differences of [0.866, 1.626].

Figure 37 shows the 95% confidence intervals of the log-transformation of defect density in unit testing for each PSP level, for both courses.

As a summary of this step, we can say that in both courses there is significant difference between all PSP levels, except between PSP2 and PSP2.1. The lack of significance between PSP2 and PSP2.1 may be because 1) the difference is too small to resolve (power of the sample), 2) the log-transformation hides the results (transforming those zeros will reduce the effect), or 3) there is no difference.

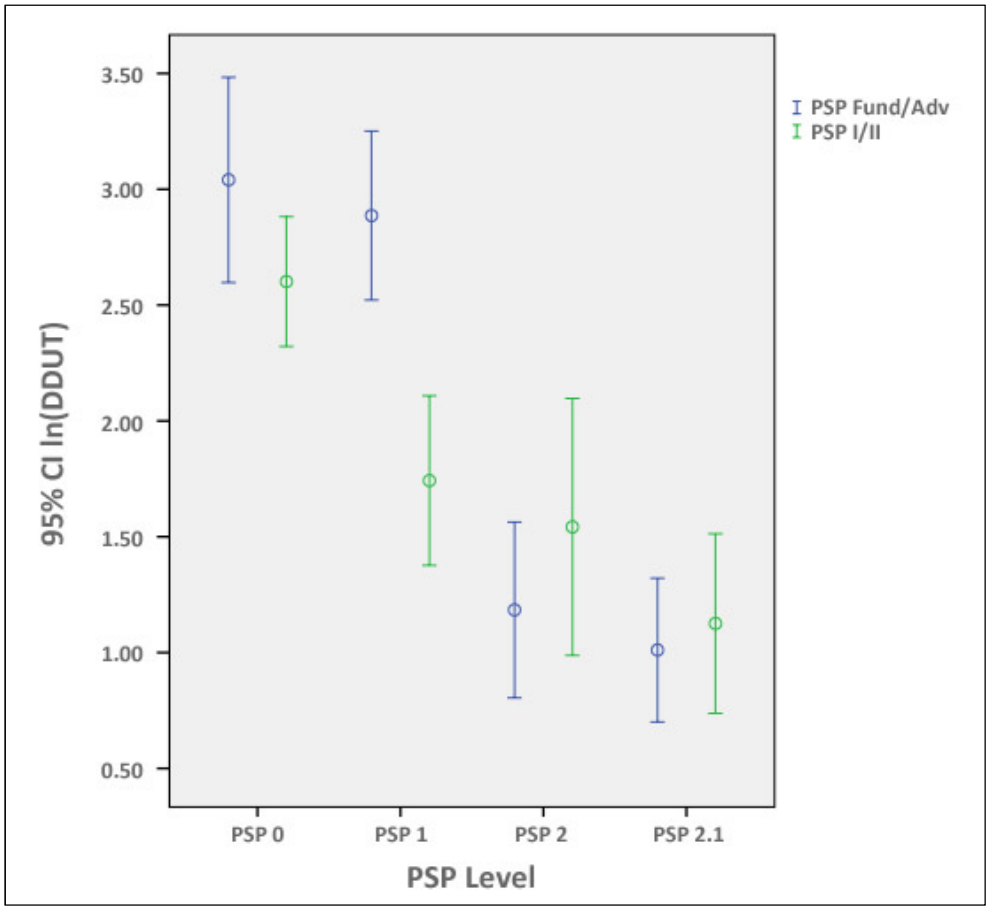


Figure 37: 95% Confidence Interval of Ln(DDUT) for each PSP Level in PSP Fund/Adv and PSP I/II

7.4 Threats to Validity and Limitations

By definition, defect density depends on the number of defects removed. But the number of defects found and removed in the test phase depends on the student's experience and on how good

the student is in doing unit testing. Therefore, we have a threat related to testing because the tests —those that are coincident with the treatment—may influence the student behavior.

According to the history, these courses were taught largely, but not entirely, at different times. Newer development environments and changes in the computer language instruction may alter subject behavior or the defect injection profile.

For a correct application of ANOVA, there is an assumption that the subjects are randomly selected for the treatments. We did not select the students; they were the ones that selected the course, and there is no precondition to do one course or another. So the random selection seems to be satisfied. But on the other hand, the students who took the PSP Advanced are more likely to go on to instruction or teaching. So, this group might respond better to the PSP instruction, and this could be seen as a threat to validity.

Even after transformation of the data, the normality assumption for ANOVA could not be satisfied. The distributions of defect density tend to be positively skewed, with long tails extending to the right and a truncated range at zero. This type of non-normal distribution is to be expected given the source of the data. There can never be a negative count for defects, so the truncation at zero is expected. In addition, we would expect many small values of defect density and relatively fewer large values. This positively skewed distribution is expected particularly when engineers (as a group) reduce the defect density of the programs as they improve their quality during the course. This is the type of data where either a logarithmic or inverse transformation can be used to create a more nearly normal distribution [Tabachnick 1989]. Based on our examination of the effects of these two types of transformations on the distribution of residuals, the logarithmic transformation was used in the confirmatory analysis.

We are comparing program assignments of two course versions as if they were identical. This can be considered a threat to validity. While the program assignments are very similar between the courses with the same programming exercise, they differ in the process elements and process used, so they are not exactly the same in both courses.

7.5 Conclusions

Previous studies of Personal Software Process [Hayes 1997, Rombach 2008] have examined the effect of the PSP on the performance of software engineers. The improvements, including product quality, were found to be statistically significant, and the observed results were considered generalizable beyond the involved participants. Those studies only considered students of the first version of the PSP course, which uses 10 program assignments and where there is a strong correlation between the program assignment and PSP level. Those studies may have some threats to external validity. In this work we try to face the generalization threat and consider the latest two course versions to see how different process introduction approaches correlate with quality results in PSP courses.

In this analysis we considered the work of 93 software engineers, who during PSP work developed eight or seven programs, depending on the course version. Each subject took the complete PSP course, either PSP for Engineers I and II or PSP Fundamentals and Advanced. We analyzed the data collected by each student to see how review and design improve the quality of a product in test.

Both courses appear to be effective in demonstrating use of design and reviews and both show reduction in defect injections. Levels achieved at end of the course are consistent with best-in-class practice. This cross-course comparison allowed us to discover that a "Hawthorne effect" is not as plausible as "gaining experience in the problem domain" or PSP techniques associated with PSP level as a causal explanation for the improvements. The strong association with PSP level suggests that learning effects are most plausible regarding mastering PSP techniques rather than general domain knowledge. This might be further examined in a future study with an analysis of phase injection and removal.

Because PSP level changes so rapidly in the PSP Fundamentals and PSP I, program number and PSP process level are tightly correlated in a way that makes separating the effects difficult. These results cannot ensure that the observed improvements are exclusively due to mastering the process techniques introduced in the PSP. We propose future analysis to obtain more generalizable results. The first approach would be a control experiment, consisting of introducing students to PSP, then remaining at PSP 1.0 through the first seven program assignments used in PSP I/II and PSP Fundamentals/Advanced. In this way, we can see how domain learning affects software quality improvement and then compare that result with the results of this study. A second approach would be an extended PSP course with at least three exercises at each PSP level. We believe that this would permit the process changes to stabilize so that we could more directly examine improvements between programs with and without process change.

In future analysis of these data from PSP Fundamentals/Advanced and PSP I/II, we will examine improvements in other dimensions, such as size estimation, effort estimation, defect yield, and productivity, to determine how effectively these courses work in those dimensions.

7.6 Acknowledgments

We thank Jim McCurley of SEI, Gabriela Mathieu, and Diego Vallespir of Universidad de la República for discussions of ANOVA analysis and suggestions of different statistical methods. We also thank the reviewers for their valuable contributions.

7.7 Author Biographies

Fernanda Grazioli
Graduate Student
Universidad de la República

Fernanda Grazioli is a graduate student at the Engineering School at the Universidad de la República, an honorary collaborator of the Software Engineering Research Group (GRIS), and a member of the Software and System Process Improvement Network in Uruguay (SPIN Uruguay).

William Nichols
Bill Nichols joined the Software Engineering Institute (SEI) in 2006 as a senior member of the technical staff and serves as a PSP instructor and TSP coach with the Team Software Process (TSP) Program. Prior to joining the SEI, Nichols led a software development team at the Bettis Laboratory near Pittsburgh, Pennsylvania, where he had been developing and maintaining nuclear engineering and scientific software for 14 years. His publication topics include the interaction patterns on software development teams, design and performance of a physics data acquisition

system, analysis and results from a particle physics experiment, and algorithm development for use in neutron diffusion programs. He has a doctorate in physics from Carnegie Mellon University.

[Batini 2006]
Batini, C. & Scannapieco, M. *Data Quality: Concepts, Methodologies and Techniques.* Springer-Verlag, 2006.

[Davis 2003]
Davis, B. N. & Mullaney, J. L. *The Team Software Process (TSP) in Practice: A Summary of Recent Results* (CMU/SEI-2003-TR-014), Software Engineering Institute, 2003. http://www.sei.cmu.edu/library/abstracts/reports/03tr014.cfm

[Hayes 1997]
Hayes, Will & Over, James. *The Personal Software Process: An Empirical Study of the Impact of PSP on Individual Engineers* (CMU/SEI-97-TR-001). Software Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/library/abstracts/reports/97tr001.cfm

[Humphrey 1995]
Humphrey, Watts S. *A Discipline for Software Engineering*. Addison-Wesley, 1995. http://www.sei.cmu.edu/library/abstracts/books/0201546108.cfm

[Kemerer 2009]
Kemerer, Chris & Paulk, Mark. "The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data." *IEEE Transactions on Software Engineering 35*, 4 (July–August 2009): 534-550.

[Lee 2002]
Lee, Y. W.; Strong, D. M.; Kahn, B. K.; & Wang, R. Y. "AIMQ: A Methodology for Information Quality Assessment." *Information & Management, 40, 2* (December 2002): 133-146.

[Mayo 1949]
Mayo, E. *Hawthorne and the Western Electric Company*. "The Social Problems of an Industrial Civilization." Routledge, 1949.

[Neely 2005]
Neely, M. P. "The Product Approach to Data Quality and Fitness for Use: A Framework for Analysis," 221–236. *Proceedings of the 10th International Conference on Information Quality* Boston, MA, November 2005. MIT Press, 2005.

[O'Hara 2010]
O'Hara, R. B.; & Kotze, D. J. "Do not log-transform count data." *Methods in Ecology and Evolution 1* (2010): 118–122.

[Paulk 2010]
Paulk, Mark C. "The Impact of Process Discipline on Personal Software Quality and Productivity." *Software Quality Professional 12*, 2 (March 2010) 15–19.

[Paulk 2006]
Paulk, Mark C. “Factors Affecting Personal Software Quality.” *CrossTalk: The Journal of Defense Software Engineering 19*, 3 (March 2006): 9–13.

[Rombach 2008]
Rombach, Dieter; Munch, Jurgen; Ocampo, Alexis; Humphrey, Watts S.; & Burton, Dan. “Teaching Disciplined Software Development.” *The Journal of Systems and Software 81*, 5 (2008): 747–763.

[Strong 1997]
Strong, D. M.; Lee, Y. W.; & Wang, R. Y. “Data Quality in Context,” *Communications of the ACM 40*, 5 (1997): 103–110.

[Tabachnick 1989]
Tabachnick, B. G. & Fidell, L. S. *Using Multivariate Statistics.* Harper Collins, 1989.

[Wang 1995]
Wang, R. Y.; Reddy, M. P.; & Kon, H. B. “Toward Quality Data: An Attribute-Based Approach.” *Decision Support Systems13,* 3–4 (March 1995)*:*349-372.

[Wohlin 1998]
Wohlin, C. & Wesslen, A. “Understanding software defect detection in the Personal Software Process,” 49–58. *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, Paderborn, Germany, November 1998. IEEE, 1998.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2012	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
TSP Symposium 2012 Proceedings	FA8721-05-C-0003

6. AUTHOR(S)

William Richard Nichols; Álvaro Tasistro, Diego Vallespir, João Pascoal Faria, Mushtaq Raza, Pedro Castro Henriques, César Duarte, Elias Fallon, Lee Gazlay, Shigeru Kusakabe, Yoichi Omori, Keijiro Araki, Fernanda Grazioli, & Silvana Moreno

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2012-SR-015

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

The 2012 TSP Symposium was organized by the Software Engineering Institute (SEI) and took place September 18–20 in St. Petersburg, FL. The goal of the TSP Symposium is to bring together practitioners and academics who share a common passion to change the world of software engineering for the better through disciplined practice. The conference theme was "Delivering Agility with Discipline." In keeping with that theme, the community contributed a variety of technical papers describing their experiences and research using the Personal Software Process (PSP) and Team Software Process (TSP). This report contains the six papers selected by the TSP Symposium Technical Program Committee. The topics include analysis of performance data from PSP, project performance outcomes in developing design systems, and extending the PSP to evaluate the effectiveness of formal methods.

14. SUBJECT TERMS	15. NUMBER OF PAGES
defect types, code defect injection, time estimation, product quality improvement	100

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL