Quality Is Free, Personal Reviews Improve Software Quality at No Cost

DIEGO VALLESPIR AND WILLIAM NICHOLS

The largest single portion of software development time is usually spent in test, and that time is dominated by finding and fixing defects and then retesting. Test time also tends to be highly variable and finds only a modest portion of the defects. However, because developers tend to repeat the same mistakes, the types of defects to be removed can be predicted. Developers can find these known types of defects using the technique of structured personal review, and the checklist-based personal review used in the Personal Software Process℠ (PSP℠) they can use for production code.

Analyzing PSP classroom data, the authors found that personal reviews can remove 60 percent or more of the defects before unit testing. Because finding and fixing the defects during a review is less expensive than fixing them during test, the review time pays for itself. Moreover, the defect density is lower going into test than without review, thus making testing effort more predictable and, more importantly, enabling testing to focus on the more difficult types of defects. Although it is tempting to review or inspect code after test, the authors' study of the economics suggests it is better to review before the product is used in any other activity. This research is based on an ongoing retrospective study conducted using data from PSP courses. The results suggest that all developers should personally review their work to improve quality at no net increase in cost.

KEY WORDS

cost of quality, empirical study, personal review, Personal Software Process

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. One can then make informed decisions about preventing defect injection or committing the necessary effort to remove the injected defects.

The Personal Software Process℠ (PSP℠) establishes a highly instrumented development process that includes a rigorous measurement framework for effort and defects (Humphrey 2005). After examining a large amount of data generated during the PSP instruction classes, the authors know how many defects are injected by developers using the PSP, when they are detected, and the effort required to remove them.

The authors have found that even using a rigorous PSP development process, nearly a quarter of all defects detected will still be present at the beginning of the unit test. Regrettably, finding and removing defects in the unit test phase requires several times as much effort as removal in earlier phases.

On the other hand, the authors also found that design and code personal reviews are highly effective in detecting and removing

defects. Fifty-three percent of all defects injected during the design phase are detected in the design review phase. The code review phase detects 62 percent of all defects injected during the code phase.

The purpose of this study is not to measure the effectiveness of PSP training, but rather to characterize where the defects are injected, where they are removed, and how much it costs to find and fix them. By examining the characteristics of defect injections, removals, and escapes, developers can learn how to define and improve their own processes, and thus make product development more effective and efficient.

The PSP is learned in a course. During the course, engineers complete exercises that require them to build programs while they progressively learn the PSP. There are eight exercises in the particular PSP course the authors analyzed. In this article, they present an analysis of the defects injected in the last three PSP programs (exercises 6, 7, and 8), which were built by engineers using the complete PSP.

The authors' analysis of the complete data available from individual defect logs shows that defects are more expensive to remove in the unit test phase than in previous phases of the process. Furthermore, they present evidence of the effectiveness of personal reviews and emphasize their importance if the aim is to obtain a quality product. They show how the defects injected escaped into each subsequent phase of the PSP and how the cost of removing them is affected by the combination of the phase injected and the phase removed.

Other articles have also been published about software quality improvements using PSP (Ferguson 1997; Hayes and Over 1997; Paulk 2006; Paulk 2010; Rombach et al. 2008; Wohlin and Wesslen 1998). These articles discuss quality in terms of defect density (defects/KLOC), as it is measured in the context of PSP. The authors' study differs from these in that it focuses on the percentage of injected and removed defects and costs and does not consider defect density.

Another difference is that the authors analyze only the final three programs of the PSP course, when the engineers have learned the complete PSP and are applying it in the programs.

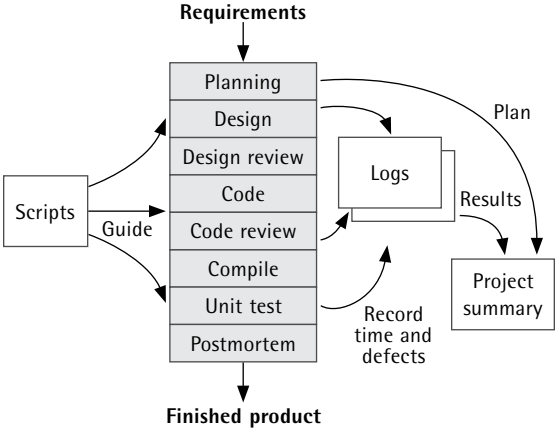
An early PSP report (Humphrey 2000) suggested that time in review should be approximately 65 percent the coding time. This was based on observed phase defect injection and removal rates. Development environments have changed somewhat in the intervening years, notably removing an explicit compile, interactive checking with IDEs, and including enhanced debugging tools. The author's analysis approach differs from that report, but will be broadly in agreement in that recommended review rates remain unchanged.

THE PERSONAL SOFTWARE PROCESS

"The Personal Software Process (PSP) is a self-improvement process that helps you to control, manage, and improve the way you work" (Humphrey 2005). This process includes phases that are completed while building the software.

For each software development phase, the PSP has scripts that help software engineers 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 (that is, remove) 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 PSP phases, scripts, logs, and project summary



Some clarification is 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 the code review, compile, and unit test phases. Once a block of code has passed into a phase, all time is logged in that phase, regardless of the developer's activity. For

example, a failure in test will require some coding and a compile, but this time is logged as the unit test phase.

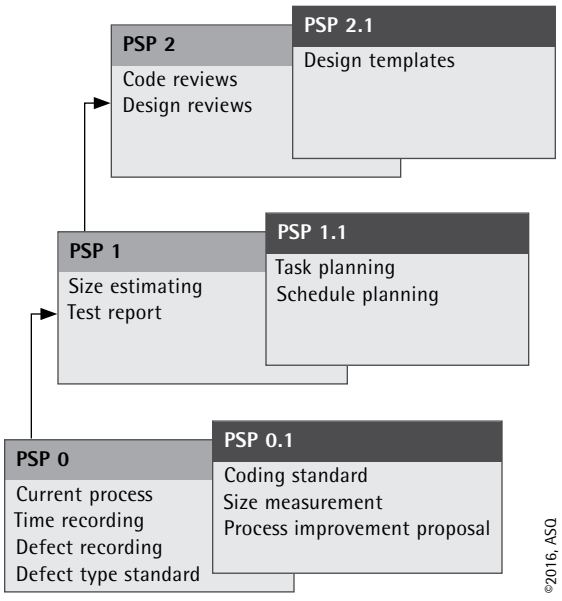
The time to find and fix a defect includes the time it takes to find it, correct it, and then verify that the correction made is right. In the design review and code review phases, the time to find a defect is zero, since finding a defect is direct in a review. However, the time to correct it and check that the correction is right depends on the complexity of the correction.

On the other hand, finding a defect is an indirect activity in both the compile and unit test phases. First, there will be a compilation error or a test case that fails. Taking that failure as a starting point (compilation or test), the cause of the defect must be found in order to make the correction and verify it.

During the PSP course, the engineers build programs while they progressively learn 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 reviews. The process changes as these elements are added. The name of each process and which elements are added in each are presented in Figure 2. The PSP 2.1 is the complete PSP process.

In PSP 2.1, students conceptualize program design 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, do unit testing.

FIGURE 2 PSP process level introduction during course



PERSONAL REVIEWS

A personal review is a way to find defects quickly by examining one's own products. The activities are similar to those performed in a peer review, but they are personally performed by the developer on his or her work product before peer inspection, testing, or even compiling. The use of interactive integrated development environments (IDEs), debuggers, and test-driven development, and the virtual disappearance of a separate compile, have led to increased interaction with the tools and less reliance on the personal review. The authors suggest, however, that while these tools reduce some costs, they hide others. While modern tools make development more interactive and engaging, failure to systematically review removes a valuable opportunity to learn from one's mistakes. Moreover, properly performed personal reviews remain a consistently effective and efficient way to remove defects early while using other modern development tools. Rather than give up the new and go back to punch cards and batch compile, the authors argue for using one of the most effective quality tools from an earlier era—the personal review. To make this case, they describe the review process and examine the economics as measured in the PSP.

To perform a review, software professionals critically examine the product to find defects. To be effective, they should follow a defined review process with input criteria, execution steps, and exit conditions. A review should use a checklist appropriate for the product. It is best to review a product after producing it and before using that product in the next step. For example, requirements can be reviewed before design, designs reviewed before coding, and code reviewed before inspection or test. Most software defects result from simple oversights or errors that are easiest to find and fix soon after producing the design or the code. If software professionals review the product too soon after producing it, they may see what they expect to see rather than the defect; if they wait too long, however, they may forget exactly what they meant. Taking a modest break before review, between an hour and a day, helps make the mistakes more visible

while software professionals still remember what they intended. Before completing the review, note the types and counts of defects and the time spent reviewing. The data will be useful later for analysis, but the authors also conjecture that the objective and prompt feedback enables self-learning, thus reducing the incidence of some future defects.

There are many ways to do a code review, but the authors recommend printing a source code listing. While one can review the code on the computer screen, there are advantages to a printout. With a printout one can mark up the paper without becoming distracted with the fix, and easily obscure portions, annotate relationships, note alternative options, draw lines to mark off sections, apply a highlighter, and so forth. The authors' experience has shown that reviews are more effective when using a paper listing and reviewing using a structured approach. An informal reading often misses simple mistakes, as the reviewer sees what he or she expects to see rather than what is actually printed.

Using a personal code review checklist, developers first select each item for review in turn and proceed to review the entire document for that item. The personal code review checklist is developed taking into account the personal defects the developers have found in their own work. It is updated regularly to focus on the few key types of defects that are most troublesome. When each action is completed, the developer checks it off the list. At the end, the developer reviews the entire checklist to ensure that he or she has checked every item. If not, the developer must go back and perform the missing actions, check them off, and again scan the list to make sure he or she did not miss anything else. In using a checklist, the following practices may be helpful:

- Take a break before review. Developers may want to start another module, class, or procedure before reviewing the one they just completed.
- Plan to review no more than 200 to 300 lines of code at a rate of no more than 200 lines of code per hour. If necessary, separate classes, modules, or procedures to get to a manageable size.
- Print out the program listing to perform the review.
- Use a review checklist tailored to the defects the developers personally inject. Describe types of defects in a way meaningful to developers so that when they see the defect, it will be apparent.
- For each item on the checklist, go through the entire document looking for only that item. If developers see other defects, mark them and move on. Only after completing the entire document should they move on to the next item.

A design review should be performed after design activities but before coding. The purpose is to ensure that the design is complete and correct before proceeding. Designs usually capture the structure of the code, distribute functionality or responsibility among components, define the interfaces, specify calling sequences, determine data structures, choose algorithms, and specify internal logic. Various representations help developers conceptualize at a more abstract level than code. These may include state charts, sequence diagrams, flowcharts, pseudocode, and class diagrams. Reviews typically use a checklist to check for complete satisfaction of requirements, correctness of logic, termination of loops, undefined state transitions, matching of the interfaces, and so forth. The checklist is supplemented with analysis tools appropriate to the specific design representations such as state transition analysis or symbolic execution.

It is possible to conduct design review after coding. Competent programmers can usually produce small products without an explicit design. Developers may find that they need to write prototype code to test performance or to better understand the problem. A prototype code is worth examining thoroughly before converting it into production code. For a simple program or module, producing a simple design should not require much time or effort but will make the review more effective. Because design defects are usually of a different character than code defects, they are more likely to escape a code review than a design-specific review. In the remainder of this paper the authors will discuss the differences between the defects found in design and code reviews and those found in test.

THE DATA SET

There are several versions of the PSP course. The authors used data from the version of the PSP course that has eight exercises from classes taught between October 2005 and January 2010 by the Software Engineering Institute (SEI) at Carnegie Mellon University or by SEI

partners, including a number of different instructors in multiple countries.

This study is limited to consider 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 these techniques are necessarily applied well because the students are in a learning process.

This constitutes a threat to the validity of this study in the sense that different (and possibly better) results can be expected when the engineers use PSP in their own working environments after taking the course, if they continue to assimilate the techniques and the elements of the process after completing the course.

A related threat is that the programming exercises and results are not representative of all real-world programming. The program exercises were designed to be challenging but also to be completed in an afternoon. The variation of program size is therefore small. Developers used their own choice programming and were instructed to use a familiar programming environment. A prior publication (Davis and Mullaney 2003) showed excellent quality results with teams trained using PSP and applying this practice in industrial settings. A report summarizing results using a larger data set will be forthcoming in future work.

The authors' data set includes data from 133 students who completed all the programming exercises of the mentioned courses. From this they 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, the authors removed data from three students. Johnson and Disney reviewed the quality of PSP data (Johnson and Disney 1999). Their analysis showed that 5 percent of the data were incorrect; however, many or most of the errors in their data were in-hand calculations of derived measures used to make estimates for the next program. For example, calculation errors were found in the defects injected per hour, defect in a phase, or in calculating the regression parameters relating the estimated size to the real size and effort for the program.

Because the authors' 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 percent) was lower than the percentage reported by Johnson and Disney, but seems reasonable and should not bias the analysis.

The authors next reduced the data set to separate programming languages with more common design and coding characteristics. As they analyzed the code defects, it seemed reasonable to consider only languages with similar characteristics that might affect code size, modularity, subroutine interfacing, and module logic.

The most common language used was Java. To increase the data set size, the authors decided to include the data generated by students who used Java, C#, C++, and C. This group of languages share similar syntax, subprogram, and data constructs and are unlike languages such as COBOL or Perl. For the simple programs produced in the PSP course, the authors judged that these were most likely to have similar modularization, interface, and data design considerations. This cut reduced their data to 94 subjects.

Because the authors' intent was to analyze defect removal, they eliminated from consideration any data for which defects were not recorded. From the 94 remaining engineers, two developers recorded no defects at all in the three programs considered. Their data set for this analysis was then reduced to 92 engineers.

The authors' analysis studied the defect injection and removal performance for individual developers rather than pooling data for an overall average. That is, they wanted to characterize the work of individual programmers, and therefore they calculated individual performance for each developer. After obtaining the performance of each developer, they calculated the mean of individual performances using only the three programs they completed using the complete PSP.

So, for this study the authors included the 92 engineers using one of several languages who also recorded removed defects. However, in several analyses, the number of engineers included varies, and in each of these cases they document the reason.

WHEN ARE THE DEFECTS INJECTED AND REMOVED?

The authors know from previous work that in PSP, most of the injected defects are injected during the code and design phases (Vallespir and Nichols 2011). In their population, almost 99 percent of the defects are injected in these two phases; the remaining 1 percent are injected in the other PSP phases (this applies to PSP, but in industrial software development, high-level design and requirement specification are other important sources of defect injections). The authors found 46.4 percent of defects were injected during the design phase and 52.4 percent were injected

during the code phase. This percentage is calculated as an unweighted average of the average percentage of defects injected in each phase by each individual in the last three programs. This differs from the average of the total defects injected by their entire population. Instead, the authors are modeling an average individual. In this population, roughly half of the defects were injected in the design phase and the other half were injected in the code phase. The standard deviation shows, however, that the variability between individuals is substantial. Nonetheless, in this study the authors focus on the average situation rather than go into greater depth on the variations.

Phases in Which Defects Are Removed, Segmented by Origin

From the 92 engineers in the authors' data set, there were four whose records of injected defects during the code phase were uncertain regarding their correctness and therefore were dismissed for this analysis. Also, eight engineers did not record defects in the code phase, so they were dismissed as well. The authors' data set for analyzing the defects injected during the code phase was, therefore, reduced to 80 engineers. The data set for analysis of the defects injected during the design phase was reduced to 83 engineers for similar reasons.

For each engineer who injected defects, the authors identified the phases in which the engineers found and removed those defects. For every phase with a removal, they determined the percentage of defects that were found and removed in that phase.

Figure 3 shows the mean for each phase in which the defects were removed. From this they learned that approximately 53 percent of the defects injected during design were found in the design review phase and that 62 percent of the defects injected during code were found in the code review phase on average. These numbers show a high percentage of defects found during review, indicating that a PSP personal review is effective in finding defects. Personal reviews improved the quality of the product that goes into the unit test phase.

Considering only the defects injected during the design phase, the code and code review phases have a similar percentage of defects removed. Approximately 10 percent of the design defects were found and removed in each of these phases. Approximately 2.5 percent of the design defects were found during the compile phase. It is likely that the defects injected during the design phase and found in the compile phase were pseudo-code defects (in PSP the pseudo-code is written during the design phase). Finally, around 25 percent of the defects were found in the unit test phase. This means that in the PSP accounting, one of every four defects injected during design escapes all phases prior to unit test.

Considering the defects injected during the code phase, the authors show that around 20 percent of the defects escape all phases prior to unit test and around 16 percent are found in compile.

Defect Removal by Phase

The PSP defines defect removal yield in terms of phase containment. A defect is counted as a defect only if it escapes a development phase or if it results from a fix injected during test and found in a test. The defect removal "process yield," or yield, is then the fraction of defects found cumulatively through some process phase divided by the defects injected prior to that phase. The "phase yield" is the fraction of defects removed by a phase divided by the defects entering that phase. When expressed as yield, process yield is implied.

To simplify the yield calculations and presentation, the authors considered only the defects injected during the most common injection phases, design and code, thus removing from consideration 1.2 percent of the defects. Using the data in Figure 3, the authors calculated the cumulative removal percentage for each phase of the PSP. In Figure 4 they present the cumulative removal percentages separately for defects injected in design and code.

FIGURE 3 Phases where the defects are found (percentage) divided by injection phase

Phase injected	Phase removed				
	DLDR	Code	CR	Comp	UT
Design	53.4	9.6	8.9	2.5	25.7
Code	-	-	62.0	16.6	21.4

FIGURE 4 Process yield in each phase of the PSP

	Process Yield by Phase				
	DLDR	Code	CR	Comp	UT
Injected in design	53.4%	63%	71.9%	74.4%	100%
Injected in code	-	-	62%	78.6%	100%

They know, of course, that unit test is not 100 percent effective and will not have 100 percent *phase yield*. Since not all the defects that escape into unit test are found in unit test, their unit test value is a lower limit. Therefore, the authors' reported percentage of the defects found in each of these phases as greater than the actual to the extent that defects escape unit testing. To correct for this, an estimate or measurement of the unit test yield in PSP will be made.

Capers Jones presents data showing that the unit test yield is about 30 percent (his data are not limited to those using the PSP). He adds that using certain methods such as design and inspections before unit test increases the yield by 5 percent (Jones 2013).

Given that the PSP process ends up in unit testing, the authors do not have PSP data for the actual unit test phase yield. They believe that in a disciplined process like the PSP it is possible to achieve a higher yield in unit test than that presented by Jones. As a working hypothesis, the authors will assume that phase yield is 50 percent. That is, half the defects that get to the unit test phase are detected, and the other half are not. This hypothesis should be studied in future work, but their conclusions are not sensitive to a reasonable range of values. Using this assumption, the authors recalculated the revised yield of each of the phases and the process yield in each phase. The results are presented in Figure 5.

The adjustment has a small effect, changing the percentage of defects injected during design to 47 percent and defects injected during code to 53 percent. These are used to calculate the adjusted yield of all the defects, which is also presented in Figure 5.

The PSP achieves a process yield of 81 percent in unit test with the assumption of a 50 percent phase yield. This, in comparison with other data obtained in the industry (Jones 2013), indicates excellent quality.

Figure 6 presents how the PSP would perform for 100 injected defects. The figure shows where the defects are injected, how many are removed in each phase, and how many defects escape into a following phase. The data are presented by defect origin (design defects [DD], code defects [CD], and total defects [TD]). This characterizes the PSP process according to the injection and removal of defects in each phase.

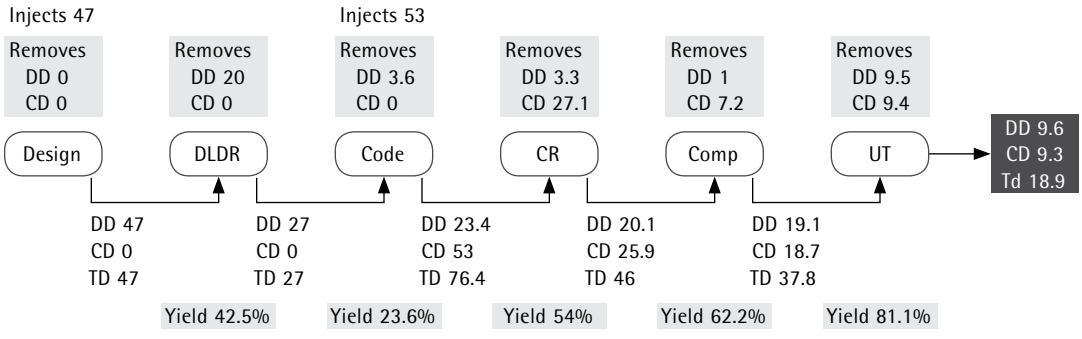
FIGURE 5 Phase yield and process yield using a 50 percent yield in UT

	DLDR	Code	CR	Comp	UT
Process yield (design phase def.)	42.5%	50.2%	57.3%	59.3%	79.7%
Phase yield (design phase def.)	42.5%	13.3%	14.3%	4.8%	50%
Process yield (code phase def.)	-	-	51.1%	64.8%	82.4%
Phase yield (code phase def.)	-	-	51.1%	28.0%	50%
Process yield (all the defects)	42.5%	23.6%	54%	62.2%	81.1%
Phase yield (all the defects)	42.5%	4.5%	39.8%	18%	50%

COST TO REMOVE DEFECTS: CONSIDERING ONLY FIND AND FIX TIME

In this section the authors analyze the cost of removing defects related to the phase where the defects are injected and where the defects are removed. The cost is calculated in minutes of developer effort to find and fix the defects. Though actual charge differs not only by country and company, but also by role (for example, analyst, designer, coder, tester, and so forth), coding defects are isolated (though not necessarily detected)

FIGURE 6 Simulation of the PSP with 100 defects injected



and fixed by code developers. "Developer minutes" is thus a proxy for developer cost to find and fix defects. The authors first present the cost of removing the defects injected during the design phase and then the defects injected during the code phase.

Cost to Remove the Defects Injected in Design

Design defects can be removed in the detailed level design review (DLDR), code, code review (CR), compile (comp), and unit test phases. For each engineer, the authors calculated the average task time of removing a design defect in each of the different phases. Because some engineers did not remove design defects in one or more phases, their sample size varied by phase. They had data from 68 engineers for DLDR, 29 each for code and CR, six for comp, and 55 for unit test. They excluded the cost of finding design defects in the comp phase because they had insufficient data for that phase.

Figure 7 shows the mean time to find and fix defects in each of the studied phases. Although the authors expected removal cost to increase in later phases, this is not what the data showed. Rather, the find and fix cost remained almost constant during DLDR, code, and CR; in fact, the cost decreased a little in these phases, though the differences were not statistically significant. Further analysis is needed to determine which of the defects found in code and CR escaped through design in an effective (as opposed to ineffective) design review phase. Regardless, any defect discovered in these phases is essentially found by an inspection process where the fix time is short because the root cause has already been identified.

FIGURE 7 Cost (in minutes) of find and fix defects injected segmented by phase removed

	DLDR	CODE	CR	Comp	UT
Injected in design	5.3	5.1	4.2	-	23.0
Injected in code			1.9	1.5	14.4

The authors are not stating here that the cost of finding and fixing a particular design defect during DLDR, code, and CR is necessarily the same. They are stating that when using PSP, the removal cost of design defects found during DLDR was approximately the same as the removal cost of those that escaped from the design phase into the code phase and those that escaped from design into CR.

On the other hand, the data showed, as expected, the average cost of finding a design defect during unit test is much higher than in the other phases, by almost a factor of five.

Cost to Remove the Defects Injected in Code

Code defects can be removed in PSP in the CR, comp, and unit test phases. For each engineer, the authors calculated the average task time of removing a code defect in each of the different phases. Because some engineers did not remove code defects in one or more phases, the authors' sample size varied by phase. They had data from 72 engineers for CR, 44 for comp, and 51 for unit test.

Figure 7 shows the mean for the find and fix time (in minutes) for code defects in each of the studied phases. Unsurprisingly, the average cost of finding code defects during unit test is much higher than in the other phases, this time by a factor of seven. Again, the authors do not claim that the cost of finding and fixing a particular code defect during unit test is seven times higher than finding and fixing the same code defect in CR or comp. Instead, they state that when using PSP, the code defects removed during unit test cost an average of seven times more than those removed in CR and comp. These are different defects.

Considering the defects injected in design and in code, the defects injected in design and removed in unit test are the most costly to remove, taking an average of 23 minutes. Defects injected in code and removed in unit test follow with an average of 14.4 minutes. This suggests that testing, even at the unit test level, is consistently more expensive than alternative verification activities.

Cost to Remove Defects: Considering Time in Phase

In this section the cost of defect removal is analyzed, but considering the entire time in phase the removal phases instead of only the find and fix time. The time in a defect removal phase (design review, code review, compile, and unit testing) is the cost associated with the application of a defect removal technique. For this

analysis, the cost of defect removal is calculated as the ratio of the defects removed in a phase and total time in phase. That is, the average cost of removal includes the time required to execute that phase, not just the fix time after discovery.

In the previous section the costs are differentiated depending on the injection phase of the defect (design or code); in this case it is impossible to make this division. The time in phase is the time engineers used in a phase, and it is impossible to tell how long it took them to find and fix defects injected during the design or code phases.

The cost of removing a defect in each of the phases for the three last programs in the course is calculated for each engineer. Then the average is calculated considering all the engineers. The ones who did not remove defects in the phases considered were not taken into account for the average. The data of 64 engineers are considered for DLDR, 75 for CR, 44 for comp, and 77 for unit test.

Figure 8 shows the average cost of removing the defects injected in each of the studied phases. The lowest average cost is in comp. On the other hand, the cost in CR is significantly lower than the unit test and DLDR, whereas in the last two it is very similar. As in the previous section, the authors are not comparing the cost of removing the same defect in each of the phases. The comparison corresponds to the cost of removing the defects that are not removed in previous phases.

FIGURE 8 Cost (in minutes) of removing defects per phase considering time in phase

	DLDR	CR	Comp.	UT	UT
Time (min.)	41.8	27.7	4.1	38.9	100%

The cost of the reviews, both of code and design, does not normally depend on the number of defects but on the size of the product to be reviewed. This aspect is not included in this work. However, it is important to know that relationship in order to analyze the cost of defect removal and the removal techniques more deeply.

DISCUSSION OF THE RESULTS

The authors found that design and code reviews are highly effective in defect removal at an early stage. Design reviews detect about half the defects, while code reviews detect about 60 percent of the defects.

This high combined effectiveness (using both reviews in the same process) controls product software quality. Assuming 50 percent of the defects that get to unit test are detected in that phase, the process that combines design review, code review, and unit test is capable of detecting 80 percent of the defects injected before integration test.

These results show that the defects injected during the design phase are more expensive to remove than those injected during code. They also show that the cost of removing defects in phases prior to unit test remains constant; some five minutes for those injected during design, and about two minutes for those injected in code. However, there is a significant quantitative increase when removing defects in unit test. The variable cost per defect increases almost five times for the defects injected in design (23 minutes) and about seven times for those injected in code (14 minutes).

Because find and fix is only the variable time, the authors also analyze the fixed cost, or the total time invested in each phase. To better analyze this situation, it is necessary to know more about the relationship between time in phase and the size of the product.

From the point of view of software engineering practice, this work highlights the economic benefits of the early defect removal activities. The data show that despite unit test removing only 23 percent of the defects, it still incurs 64 percent of the find and fix cost. Relying on unit testing as the first, best, and only method of software verification not only increases the cost of a software project, but also increases the number of escaped defects.

From the point of view of practice, the authors emphasize the importance of both preventing the injection of defects and being able to detect them prior to the unit test phase. That is, they advocate transferring costs of quality to phases where defect removal is significantly less expensive. The developer has several options, including improving the reviews, improving the design itself, investing more time in the reviews, and, in a team environment, introducing inspections (Fagan 1976). At present, the authors are researching the use of design by contract (Meyer 1992) with the PSP (Moreno, Tasistro, and Vallespir 2012) to help prevent or remove a greater number of design and code defects prior to unit test with less overall cost.

Software engineers must be aware that the techniques, benefits, and importance of inspections have been reported in the literature since the well-known article by Fagan (1976). This article presents the costs associated with the removal of defects in personal review

and in unit test using PSP data. The authors' contribution in this work is reporting empirical data demonstrating the similar benefits of the personal review. Nonetheless, much of the software industry bases its verification strategies on software testing and tools. Testing and tools serve important roles, but should supplement rather than replace effective and efficient human reviews. Finally, they emphasize the importance of having multiple defect filters before reaching unit test as a way of reducing the costs of the development process and producing a better software product.

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, the fact that students are still going through the learning process, and the nature of the exercises.

Because the PSP life cycle begins in the detailed design phase and ends in the unit test phase, the authors 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 defects found 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. PSP exercise code is not intended to be "bullet proof" or production ready. This is most likely to affect the rigor of the unit test. Students often run only the minimum tests specified. This likely leads to fewer defects being found and higher overall development rates. For example, coding rates are typically much higher than found in industry. Also excluded is the production practice of peer inspections.

A third consideration is that the students using PSP are still learning design and personal review techniques. The results after gaining experience may differ from those found during the course. Finally, the problems, though modestly challenging, do not represent a broad range of development problems.

CONCLUSIONS

In this analysis, the authors considered the work of 92 software engineers who, during PSP course work, developed programs in the Java, C, C#, or C++ programming languages. They focused their analysis on the defects injected during the design and code phases.

They found that defects were injected roughly equally in the design and code phases; that is, around half of the defects were injected in design and around half in code. About half of the defects injected in design were discovered early (in the DLDR phase). However, 25 percent of these defects were discovered late (in the unit test phase). About 60 percent of the defects injected in the code phase were also discovered early (in the CR phase). But, again, a number of the defects (21 percent) were discovered late (in the unit test phase).

Defects discovered in unit test are more expensive to remove than those removed in earlier phases. The defects injected in design and removed in unit test are almost five times more expensive to remove than the defects removed in DLDR. The defects injected in code and removed in unit test are almost seven times more expensive than the defects removed in CR. This is not only a warning to PSP's users, it is also a warning to any software developer: Do not perform unit test before reviewing the detailed design and code.

While this analysis provided insights into the injection and removal profile of defects with greater specificity than previously possible, a larger data set would allow the authors 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 them to analyze improvement opportunities to achieve better process yields.

ACKNOWLEDGMENTS

This work was partially funded by the Programa de Desarrollo de las Ciencias Básicas (PEDECIBA), Uruguay. This material is based upon work funded and supported by TSP and PSP cost recovery 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 sponsored by the U.S. Department of Defense.

REFERENCES

Davis, N., and J. Mullaney. 2003. The Team Software Process (TSP) in practice: A summary of recent results. *SEI Technical Report.* CMU/SEI-2003-TR-014. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Ferguson, P., W. S. Humphrey, S. Khajenoori, S. Macke, and A. Matvya. 1997. Results of applying the Personal Software Process. *Computer* 30, no. 5:24-31.

Hayes, W., and J. Over. 1997. The Personal Software Process: An empirical study of the impact of PSP on individual engineers. SEI Technical Report. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Humphrey, W. S. 2000. The Personal Software Process (PSP) SM. SEI Technical Report, CMU/SEI-2000-TR-022. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Humphrey, W. S. 2005. *PSP: A self-improvement process for software engineers*. Boston, MA: Addison-Wesley.

Humphrey, W. S. 2006. *TSP: Coaching development teams*. Boston, MA: Addison-Wesley.

Johnson, P. M., and A. M. Disney. 1999. A critical analysis of PSP data quality: Results from a case study. *Empirical Software Engineering* 4, no. 4:317-349.

Jones, C. 2013. Software defects origin and removal methods. Draft 7.0.

Meyer, B. 1992. Applying design by contract. *Computer* 25, no. 10:40-51.

Moreno, S., Á. Tasistro, and D. Vallespir. 2012. PSPDC: An adaptation of the PSP to incorporate verified design by contract. In *Proceedings TSP Symposium 2012*, St. Petersburg, FL, 41-50.

Paulk, M. C. 2006. Factors affecting personal software quality. *CrossTalk: The Journal of Defense Software Engineering* 19, no. 3:9-13.

Paulk, M. C. 2010. The impact of process discipline on personal software quality and productivity. *Software Quality Professional* 12, no. 2:15-19.

Rombach, D., J. Munch, A. Ocampo, W. S. Humphrey, and D. Burton. 2008. Teaching disciplined software development. *The Journal of Systems and Software* 81, no. 5:747-763.

Vallespir, D., and W. Nichols. 2011. Analysis of design defects injection and removal in PSP. In *Proceedings TSP Symposium 2011*, Atlanta, GA, 19-25.

Wohlin, C., and A. Wesslen. 1998. Understanding software defect detection in the Personal Software Process. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, 49-58.

BIOGRAPHIES

Diego Vallespir is the director of the Computer Science Institute at the Engineering School of the Universidad de la República (UdelaR), director of the Informatics Professional Postgraduate Center at UdelaR, director of the Software Engineering Research Group (GrIS) at UdelaR, and a researcher at PEDECIBA-Informatics. He holds an engineering degree in computer science, a master's degree in computer science, and a doctorate in computer science, all obtained from UdelaR. He has published several articles in international conference proceedings. His main research topics are empirical software engineering, software process, software techniques, and software engineering education. He can be reached by email at dvallesp@fing.edu.uy.

William Nichols is a senior member of the technical staff in the Software Solutions Division of the Software Engineering Institute (SEI) at Carnegie Mellon University. His current work focuses on software process measurement, project management, quality, security, and improving development team performance. During his tenure at the SEI, Nichols has worked extensively with the Team Software Process (TSP) initiative, where he currently serves as a Personal Software Process (PSP) instructor and as a TSP mentor coach. Prior to joining the SEI, Nichols earned a doctorate in physics from Carnegie Mellon University and later led a team developing nuclear reactor analysis software. Nichols is a senior member of IEEE and a member of ACM.