The Impact of Process Discipline on Personal Software Quality and Productivity

MARK C. PAULK
Carnegie Mellon University

One of the underlying assumptions of the software process movement is that increasing process discipline—the consistency with which one implements best practices—improves both the performance of developers and of the software products they build. This assumption underlies models and standards such as the Capability Maturity Model. Exactly what "improves" means depends on the business context, but typically it refers to productivity and quality. The Personal Software Process℠ (PSP℠) can be used to demonstrate the impact of process discipline in a rigorous statistical sense. PSP can be used to show higher productivity and quality following the adoption of disciplined processes, but it also illustrates some of the challenges in defining these concepts in a useful way.

Key words

Personal Software Process, productivity, PSP, quality, software process

SQP References

Integrating Improvement Initiatives: Connecting Six Sigma for Software, CMMI, Personal Software Process, and Team Software Process
vol. 5, issue 4
Gary A. Gack and Kyle Robison

Integrating PSPSM, TSPSM & Six-Sigma
vol. 6, issue 4
Steve Janiszewski and Ellen George

INTRODUCTION

One of the underlying assumptions of the software process movement is that increasing process discipline—the consistency with which one implements "best practices"—improves both the performance of developers and the software products they build. It can be difficult to separate these two since they are intricately linked by cause-and-effect relationships, yet one frequently does so because both concepts are complex and multidimensional. While process discipline does not guarantee a successful project, it does increase its likelihood (El Emam 2005).

The belief that process discipline adds value underlies models and standards such as the Capability Maturity Model for Software (Paulk et al. 1995) and CMM Integration (Chrissis 2006). Exactly what "improves" means depends on the business context, but typically it refers to productivity and quality. The Personal Software Process (PSP) can be used to demonstrate the impact of process discipline in a rigorous statistical sense (Humphrey 1995). PSP can be used to show higher productivity and quality following the adoption of disciplined processes, but it also illustrates some of the challenges in defining these concepts in a useful way. Watts Humphrey successfully applied the principles of process discipline in the Software CMM, the Team Software Process

(TSP) (Humphrey 1999), and PSP—and many studies have observed the impact of these principles on quality and productivity, respectively, for the organization, the team (or project), and the individual (El Emam 2005; Humphrey 2001; Humphrey 2002).

The PSP incrementally applies process discipline and quantitative management to the work of the individual software professional in a classroom setting (Humphrey 1995). There are four PSP major processes: PSP0, PSP1, PSP2, and PSP3. Each process builds on the prior process by adding engineering or management activities. Incrementally adding techniques allows the developer to analyze the impact of the new techniques on his or her individual performance. There are usually 10 assignments. PSP data are well suited for use in research, as many of the factors perturbing project performance and adding "noise" to research data, such as requirements volatility and teamwork issues, are either controlled for or eliminated in PSP. A variety of programming languages are used by PSP students, but unless otherwise noted, only programs written in C were considered (to remove a possible confounding factor) for a data set with 2435 observations.

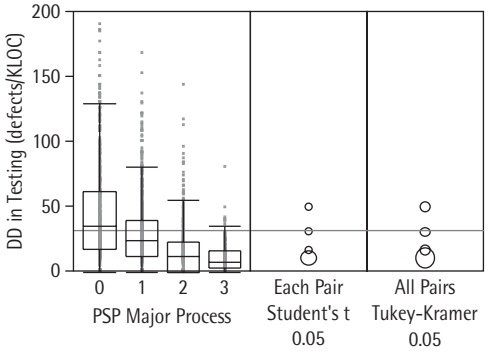
IMPACT ON QUALITY

A simple analysis of quality in PSP uses defect density, which is readily available in the PSP data. Industry products frequently use the number of defects found in the first year (or six months or some other period of time) after release as a measure of quality. Since PSP is taught in the classroom, there is no "field use" of the products. Defect density in testing is a reasonable choice, and this simple analysis is shown in Figure 1, using a box-and-whisker plot, which demonstrates a measurable and statistically significant improvement in software quality over the four PSP processes. Quality improved by 79 percent, and variability decreased by 81 percent.

Caveats abound in this analysis. While defect density may be a common quality measure, customers care little about faults—they care about failures. Faults may lie dormant for years without affecting the normal use of the software. Even quality measures such as mean-time-to-failure are flawed, however, since customers care about many attributes of the software (and the customer is the ultimate judge of quality). Garvin, for example, identified nine dimensions of quality (performance, features, functionality, safety, conformance, reliability, durability, service, and aesthetics), suggesting the underlying complexity of "total quality" (Garvin 1987). Software quality characteristics listed in ISO 9126 include functionality, reliability, usability, efficiency, maintainability, and portability (Jung 2004). Unfortunately, it is only possible to analyze defect density in the PSP context and cannot consider this larger context. Even here, however, the complexities can be daunting.

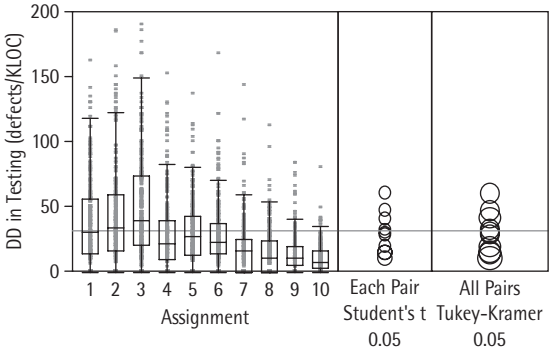
One question that may occur is whether there are differences between the problems assigned in the PSP classes that might affect the analysis. As can be observed in Figure 2, the improvement trend remains when examined per assignment rather than per process, albeit at a finer degree of granularity.

FIGURE 1 Improving quality in PSP



In a box-and-whisker plot, the "box" is drawn at the 25 and 75 percentiles in the data set, with the median as the central line. For this variant of box-and-whisker plot, the "whiskers" represent 1.5 times the interquartile range and can be used to identify outliers. The line running across the entire graph is the grand average for the whole data set. The Each Pair Student's t and All-Pairs Tukey-Kramer honest significant difference tests provide, respectively, liberal and conservative comparison tests; if the comparison circles do not overlap, or their external angle of intersection is less than 90 degrees, then you can conclude that the means of the different groups are significantly different at the given confidence level (α=0.05).

FIGURE 2 Improving quality across PSP assignments



A second question that may occur is whether one would get any better insight by looking at the number of defects rather than the defect density, if the size of the assignments are relatively the same. As can be seen in Figure 3, the sizes of the last five assignments appear to be larger than the first five, but the variability in the program sizes reinforces the poorness of lines of code (LOC) as a size measure (Schofield 2005). Since the problems are identical for all students and the programming language is constant, the differences must lie in the solutions chosen by each programmer.

Despite these concerns about the size measure being used to normalize the defect numbers, one can see in Figure 4 that the number of defects generally decreases across the PSP assignments despite the observation that the number of lines of code is increasing at the same time. This reinforces the belief that the process discipline in PSP is improving quality.

One might also ask what factors other than the process affect software quality. Among the factors that one can examine for PSP are the experience and educational background of the programmer, neither of which were found to affect quality (Paulk 2006). If one considered the larger data set with multiple programming languages, language is not a factor. Programmer ability is a factor, as shown in Figure 5.

Programmer ability was assigned based on performance on the first three PSP assignments, and the programmers were divided into four levels. As can be seen in Figure 5, taken from a repeated measures model of the data (Paulk 2005), the top performers (TQ) remain consistently better than the bottom (BQ) performers (and the middle two quarters, designated B M2 and T M2 maintain their relative positions also). The top-quarter students improved their software quality by a factor more than two, and the bottom-quarter students improved theirs by a factor more than four.

Note that programmer ability can be measured in many different ways. The chosen way emphasizes the quality of the software in testing, which could be caused by injecting few defects (high-quality development) or identifying and removing defects effectively (high-quality reviews). This analysis does not try to separate these two causes.

FIGURE 3 Size (LOC) differences in PSP assignments

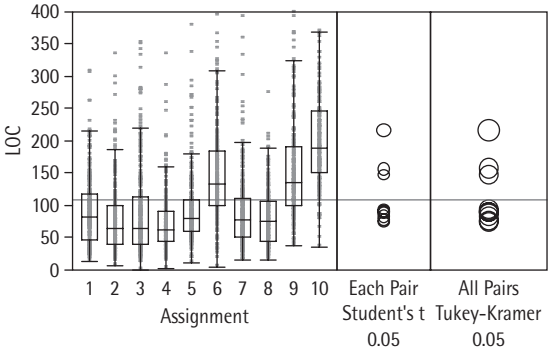
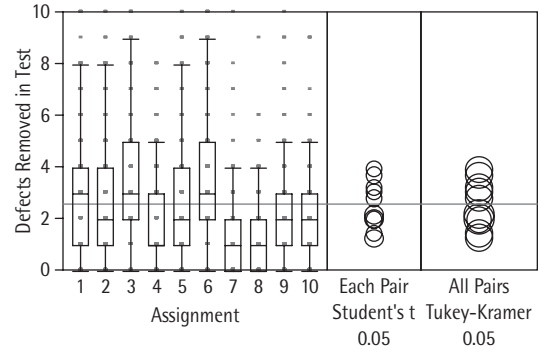


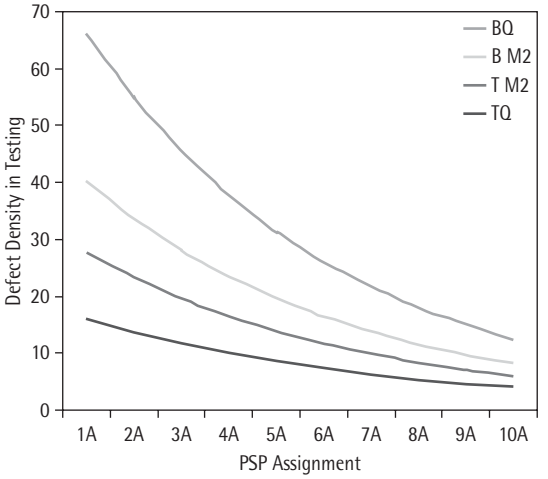
FIGURE 4 Number of defects found in testing across PSP assignments



IMPACT ON PRODUCTIVITY

A similar analysis can be performed for productivity (the ratio of output—what is produced—to resources used) as measured by LOC per hour. As shown in Figure 6, productivity increases by 12 percent across PSP

FIGURE 5 Quality trends by programmer ability



processes, with variability decreasing by 11 percent (and a statistically significant difference between PSP0 and PSP3). Whether this increase is practically significant is left for the reader to judge. In many environments, factors such as requirements volatility seem likely to swamp this small increase, but in the controlled environment of the PSP classroom, the effect is observable.

The caveats on this analysis are even greater than those on the quality analysis. LOC per hour is a horrible measure of productivity. It is unclear, however, that alternatives such as function points, requirements, or user stories per hour are better, although all four (and others) have been used for software projects (Kitchenham 2004). One alternative would be to consider the number of hours per assignment, as shown in Figure 7. While this measures productivity for each *problem* assigned, it does not allow for differences in the *solutions* chosen by each programmer (as shown earlier in Figure 3).

Years of experience and education were not observed to affect productivity. For the full PSP data set, C++ and Java were observed to be "more productive" than C and visual basic when measured by LOC per hour, but when looking at the effort spent on the problem, as shown in Figure 8 for assignment 10, there was no difference between programming languages.

When measured by LOC per hour, programmer ability was found to affect productivity. Furthermore, productivity was found to improve with quality (as suggested by both improving across PSP assignments).

FIGURE 6 Improving productivity in PSP

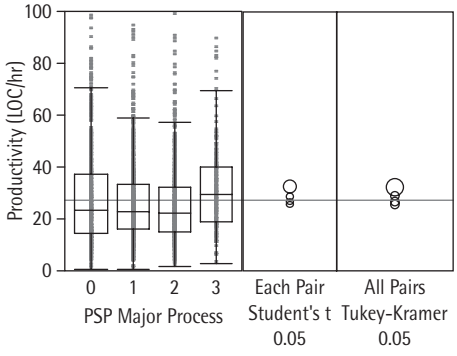
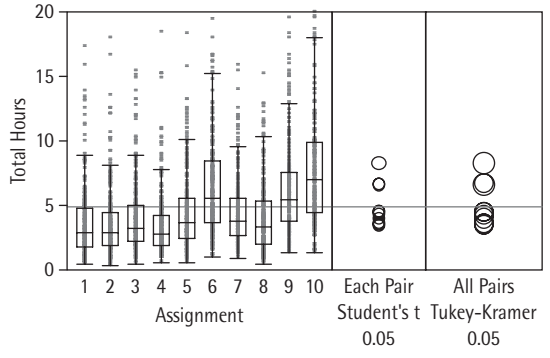


FIGURE 7 Effort across PSP assignments



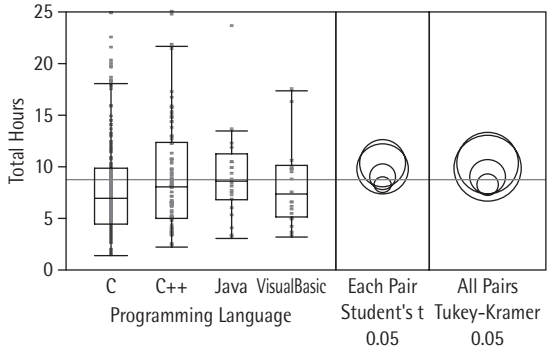
CONCLUDING REMARKS

The results reported in this article replicate those of a number of prior studies of PSP (Hayes 1997; Wesslen 2000; Humphrey 2001), but the author has focused on some of the challenges in interpreting the results in a meaningful way. Concerns are sometimes raised as to whether data from assignments done by students can be generalized to industry projects. PSP classes are frequently taught in an industry rather than an academic setting, however, and the developers in this sample reported up to 34 years of experience, with a median of seven years of experience. The PSP students therefore look more like typical industry developers than typical computer science students.

The more important consideration for PSP is the degree to which PSP assignments are comparable to industry projects. While each assignment may be comparable to a work package in a real-world project, PSP assignments do not address issues of requirements ambiguity and volatility or product integration, both of which are major issues in industry—and areas where experience and education may be crucial to success. PSP provides a foundation for TSP, and TSP projects have also been observed to have significantly improved quality and productivity (Davis 2003).

Perhaps the most challenging observation highlighted by this analysis, however, is one that is a general problem

FIGURE 8 Differences in effort by programming language (assignment 10)



for the software industry. How can one measure productivity and quality in a useful way? As demonstrated here, common measures such as defect density and LOC per hour are badly flawed. More potentially useful measures, such as rework (the percent of time spent repairing defects), are not as broadly known or used. They may perhaps be useful, especially since significantly superior alternatives are difficult to identify and implement, but any conclusions based on analyses such as these two must be thoughtfully considered before action is taken. Context is always crucial in evidence-based management.

A reviewer suggested that an equally challenging question is why PSP is not used more frequently if it is so successful. Unfortunately, many best practices in software engineering are not as widely adopted as the evidence would recommend. For example, the research supporting the effectiveness and efficiency of inspections is overwhelming, yet how many software organizations have systematically adopted any form of peer review, much less the rigor of inspections? Researchers can gather the evidence to drive informed decision making, and as teachers and consultants can advocate the adoption of best practices, but the software engineering discipline remains a young one.

References

Chrissis, M. B., M. D. Konrad, and S. Shrum. 2006. *CMMI: Guidelines for process integration and product improvement*, second edition. Boston: Addison-Wesley.

Davis, N., and J. Mullaney. 2003. The team software process (TSP) in practice: A summary of recent results (CMU/SEI-2003-TR-014). Pittsburgh: Carnegie Mellon University, Software Engineering Institute.

El Emam, K. 2005. *The ROI from software quality*. Boca Raton, Fla.: Auerbach Publications.

Garvin, D. A. 1987. Competing on the eight dimensions of quality. *Harvard Business Review* 65, no. 6 (November/December):101-109.

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

Humphrey, W. S. 1995. *A discipline for software engineering*. Boston: Addison-Wesley.

Humphrey, W. S. 1999. *Introduction to the team software process*. Boston: Addison-Wesley.

Humphrey, W. S. 2001. *Winning with software*. Boston: Addison-Wesley.

Humphrey, W. S. 2002. Three process perspectives: Organizations, teams, and people. *Annals of Software Engineering* 4:39-72.

Jung, H. W., S. G. Kim, and C. S. Chung. 2004. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software* 21, no. 5 (September/October):88-92.

Kitchenham, B., and E. Mendes. 2004. Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering* 30, no. 12 (December):1023-1035.

Paulk, M. C., C. V. Weber, B. Curtis, and M. B. Chrissis. 1995. *The Capability Maturity Model: Guidelines for improving the software process*. Boston: Addison-Wesley.

Paulk, M. C. 2005. An empirical study of process discipline and software quality. Ph.D. diss., University of Pittsburgh.

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

Schofield, J. 2005. The statistically unreliable nature of lines of code. *Crosstalk: The Journal of Defense Software Engineering* 18, no. 4 (April): 29-33.

Wesslen, A. 2000. A replicated empirical study of the impact of the methods in the PSP on individual engineers. *Empirical Software Engineering* 5, no. 2 (June):93-123.

Biography

Mark Paulk is a senior systems scientist in the Institute for Software Research at Carnegie Mellon University in Pittsburgh. He researches and teaches on best practices for software engineering and service management. This includes empirical research and case studies of best practice, with an emphasis on high maturity, agile methods, measurement, and statistical thinking. From 1987 to 2002, he was with the Software Engineering Institute at Carnegie Mellon, where he led the work on the Capability Maturity Model for Software. He was co-project editor of ISO/IEC 15504:2 (Process Assessment: Best Practices Guideline) from 1992-1995. Paulk received his doctorate in industrial engineering from the University of Pittsburgh, his master's degree in computer science from Vanderbilt University; and his bachelor's degree in mathematics and computer science from the University of Alabama in Huntsville. He is a senior member of the IEEE, a Senior member of ASQ, and an ASQ Certified Software Quality Engineer. His Web page is http://www.cs.cmu.edu/~mcp/ and his e-mail address is mcp@cs.cmu.edu.