Factors Affecting Personal Software Quality

Dr. Mark C. Paulk
Carnegie Mellon University

Understanding the factors that influence software quality is crucial to the continuing maturation of the software industry. An improved understanding of software quality drivers will help software engineers and managers make more informed decisions in controlling and improving the software process. Data from the Personal Software Process℠ provides insight into interpersonal differences between competent professionals as increasingly disciplined processes are adopted. Program size, (empirically measured) programmer ability, and disciplined processes significantly affect software quality. Factors frequently used as surrogates for programmer ability, e.g., years of experience, and technology, e.g., programming language, do not significantly impact software quality, although they may affect other important software attributes such as productivity. An understanding of these factors may help managers implement practices that support high-quality software.

The research reported in this article was part of a larger investigation into the relationship between process discipline and software quality [1]. It focuses on the product, technology, and programmer ability factors that may affect software quality in addition to the process factors. My research confirms that a disciplined process affects software quality. It also confirms that programmer ability affects software quality and, more importantly, shows that even top performers can improve their performance by a factor of about 2X by following disciplined processes. Commonly used surrogates for ability such as seniority and academic credentials are, however, likely to be ineffective measures.

The research uses data from the Personal Software Process℠ (PSP℠), which applies process discipline to the work of the individual software professional in a classroom setting. PSP is taught as a one-semester university course at several universities or as a multi-week industry training course. It typically involves 10 programming assignments, using increasingly sophisticated processes [2]. The life-cycle processes for PSP are planning, designing, coding, compiling, testing, and a post-mortem activity for learning. The primary development processes are designing and coding, since there is no requirements analysis step.

When discussing *quality* in the software industry, *defects* are the common indicator, although software quality characteristics include functionality, reliability, usability, efficiency, maintainability, and portability. Although other aspects of quality are important, software quality is measured as *defect density in testing* in the analyses described in this article.

Potential explanatory variables identified in prior research can be divided into categories related to the product and application domain, the technologies used, the software engineering processes followed, and the ability of the individuals doing the work. Quality issues related to the customer requirements are outside the scope of this research. Although requirements volatility is a significant quality concern in software projects, requirements volatility is not an issue for PSP.

Many explanatory factors for software quality from models such as COQUALMO [3] are out of scope for this research because they address project and team issues that are not relevant to the PSP environment. This highlights the challenges in generalizing PSP results to software work in general, but factors important for individual performance should also be important for teams, projects, and organizations.

> *"My research confirms ... that programmer ability affects software quality and, more importantly, shows that even top performers can improve their performance by a factor of about 2X by following disciplined processes."*

There are four PSP major processes – PSP0 to PSP3 – with minor variants for the first three (PSP3 can also be considered a minor extension of PSP2). Each level builds on the prior level by adding a few engineering or management activities. This minimizes the impact of process change on the engineer, who needs only to adapt the new techniques into an existing baseline of practices. Design and code reviews are introduced in assignment No. 7. Design and code reviews are personal reviews conducted by an engineer on his or her own design or code. They are intended to help engineers achieve 100 percent yield: All defects are removed before compiling the program. Design templates are introduced in assignment No. 9. The design templates are for functional specifications, state specifications, logic specifications, and operational scenarios.

PSP students are asked to measure and record three basic types of data: time (effort), defects, and size (lines of code [LOC]). All other PSP measures are derived from these three basic measures.

Data analyzed in this research included the data used in the Hayes and Over study [4], as well as additional data collected through 2001. These rich data sets allow sophisticated statistical analyses ranging from simple regression models to multiple regression models to mixed models that incorporate random effects and repeated measures. The conclusions reported are based on consistent results for multiple data sets. The data sets are usually split by assignment 9A or 10A (to remove potential differences in problem complexity and ensure a relatively mature process) and by programming language (to remove technology differences), both including and excluding outliers. A data set could therefore be described as (9A, C, No Outliers). In some cases, e.g., when investigating the impact of programming language used, different data splits are used as appropriate.

Confirming PSP Quality Trends

Process maturity is a generic concept that

℠ Personal Software Process and PSP are service marks of Carnegie Mellon University.

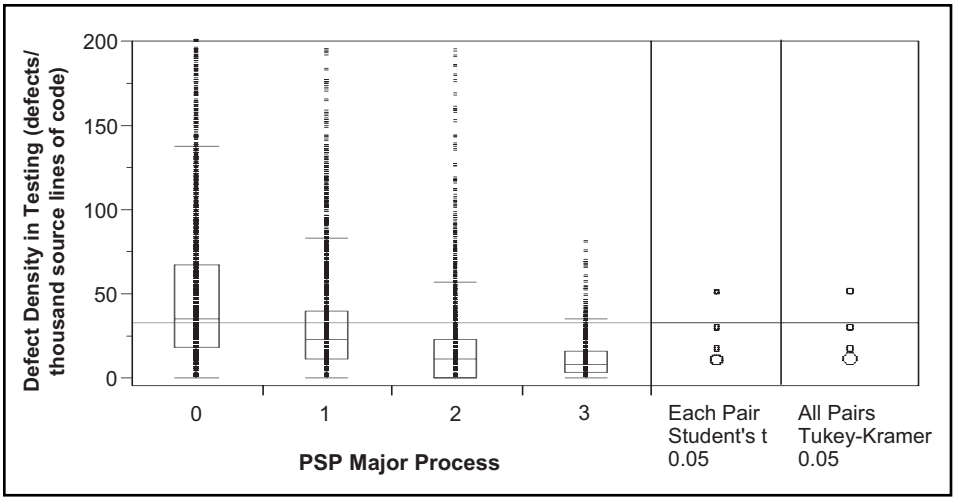


Figure 1: *Trends in Software Quality*

can be considered *low* for PSP0 and *high* for PSP3. It is interesting to note that in COQUALMO, *process maturity* has the highest impact of all factors on defect injection. As can be observed in Figure 1, which shows the differences in *defect density in testing* for each PSP *major process*, the software quality trend across the PSP's major processes is apparent, confirming prior analyses.

The comparison circles in the right part of Figure 1 indicate that the means for each of the *PSP major processes* are significantly different. The *Each Pair*, *Student's t*, and the *All Pairs*, *Tukey-Kramer honest significant difference* are separate tests of the hypothesis that the *defect density in testing* for the *PSP major processes* is significantly different. When there is no overlap of the comparison circles, as in this case, there is a significant difference, and we can conclude that the *defect density in testing* for *PSP0 > PSP1 > PSP2 > PSP3*. The differences between the PSP major processes are both statistically and practically significant. Differences greater than 4X show a decrease in *defect density in testing* of more than 75 percent over the course of PSP. This is a quality improvement that would be of interest and value to most software professionals.

These data include some outliers. Excluding outliers without causal analysis, as well as including them when they are atypical, can skew results. The analyses reported in this article are performed both with and without outliers, using inter-quartile limits (limits set at 1.5 times the inter-quartile range beyond the 25 percent and 75 percent quartiles) to identify outliers.

Program Size

Problem complexity would not appear to be a significant factor in PSP, given the relative simplicity of the assignments; programs smaller than 10,000 LOC are usually considered simple programs. S*olution complexity* is measured by new and changed LOC. A student's preferred style in optimizing memory space, speed, reliability (e.g., exception handlers), generality, reuse, etc., can lead to radically different solution complexities. Since PSP does not impose performance requirements, students have significant latitude in how they solve the problems – latitude that is available in many industry contexts as well. Using defect density as a quality measure should normalize these differences in solution complexity.

Since defect prediction models typically use *program size* as a predictor variable (and many use it as the only predictor variable), this variable is expected to be significant. *Program size* was shown to be a statistically significant effect on *defect density in testing* in all of the four data sets analyzed including outliers, but in only two of the four data sets excluding outliers. The preponderance of the evidence indicates that *program size* is related to software quality, although more weakly for PSP than for many prior defect prediction models. These somewhat atypical results highlight the impact of individual differences on software quality, although individual differences may be smoothed out by team performance on a project.

Programmer Ability

Programmer/analyst capability is difficult to objectively determine, but the PSP data provide a direct and objective measure of programmer ability. This is illustrated in Figure 2, where *defect density in testing* was averaged for the first three assignments and used to identify the top, middle two, and bottom quartiles for student performance. Note that using defect density as the measure for ability means that decreasing value of the measure corresponds to increasing ability, which means that the *top quartile* of programmers is at the bottom of the graph. *Programmer ability*, as measured by this variable, was shown to be statistically significant for every data set, including and excluding outliers.

The students who were the top-quartile performers on the first three assignments tended to remain top performers (with the smallest *defect density in testing*) in the later assignments, the middle performers tended to remain in the middle, and the bottom-quartile performers (with the largest *defect density in testing*) tended to remain at the bottom. (A spike in the data for assignment No. 3 is consistently observed for all measures, suggesting that assignment No. 3 is somewhat more complex than the norm for the PSP assignments.)

While top-quartile students performed better than those in the bottom quartile on average, a disciplined process leads to significantly better performance for the bottom-quartile students, and even the top-quartile students improved markedly. Throughout the PSP course, top-quartile students improved their software quality by a factor of more than two, and bottom-quartile students improved theirs by a factor of more than four. Variation in performance within each quartile also decreased markedly.

Potentially Confounding Variables

A number of variables could affect the analyses if not appropriately addressed. Potential confounding variables include those associated with instructor differences; surrogates for ability such as credentials, experience, recent experience, and breadth of experience; and technology factors such as the programming language used.

PSP Classes

There are two situations that could cause systemic differences across PSP classes: (1) changes in the teaching materials used in the PSP class, or (2) differences between instructors. The possibility of a trend due to systemic changes in the student population does not appear likely since there are no known reasons for a systemic change in the student population for PSP, although sporadic cases of exceptionally poorly prepared or well-prepared classes could occur. The PSP class has been based on the text "A

Discipline for Software Engineering" since its publication in 1995; prior classes used drafts of the manuscript. All PSP instructors are qualified and authorized by the Software Engineering Institute, and instruction is typically done by teams of instructors. There would not appear to be any reason for changes in PSP class performance over time although there might be cases where particular classes might have significantly different results because of special causes of variation.

Combining a small number of finishing students with one or two students struggling to finish assignment No. 9 and/or No. 10 leads to the occasional atypical class, which is not unexpected given the large number of classes being analyzed. There does not appear to be a statistically or practically significant trend over time, and it seems reasonable to conclude that, in general, PSP classes are relatively stable learning environments, although some students may have trouble on some assignments.

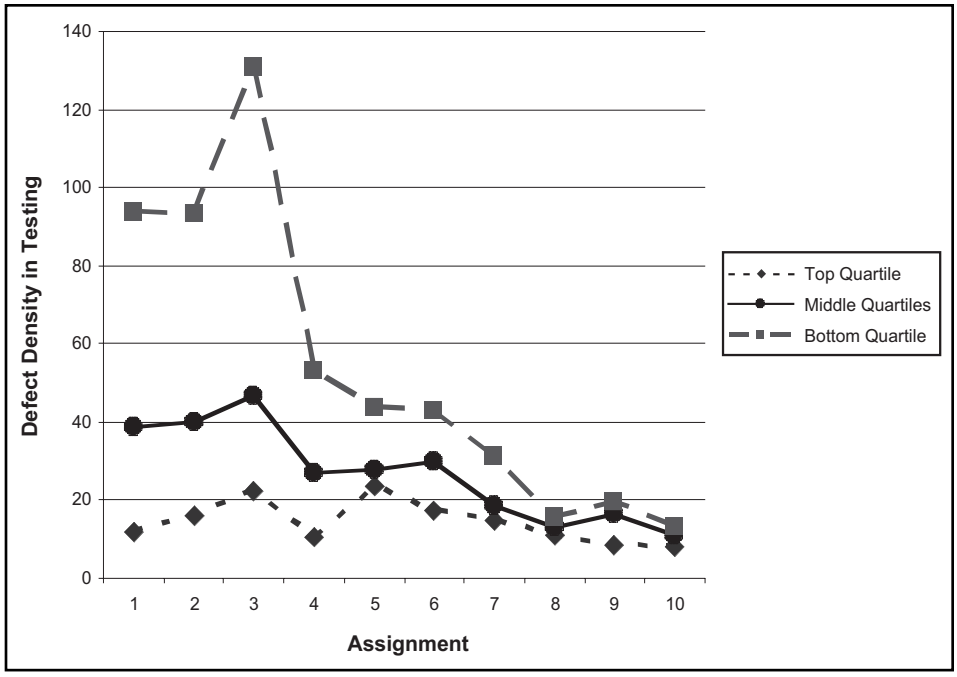


Figure 2: *Trends for Programmer Quartiles*

Finishing the Course (Or Not)

If there is a difference in performance on the early assignments between the people finishing the course and those who do not, the student population may be different from the general programming population. Finishing the course was shown to have a statistically significant effect on *defect density in testing* in only one of the four data sets analyzed, including outliers, and in none of the data sets excluding outliers. This suggests that people finishing the PSP course are reasonably typical of programmers who choose to take the PSP course. This does not, however, necessarily indicate that PSP students are representative of the population of software professionals in general.

Highest Degree Attained

As illustrated in Figure 3, the *highest degree attained* (doctorate [Ph.D.], Master of Science [MS], Bachelor of Science [BS], or Bachelor of Engineering [BE]) was not shown to be a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers. The overlap in both sets of comparison circles graphically show that *defect density in testing* for students with a doctorate overlaps that of students with a bachelor's or master's degree.

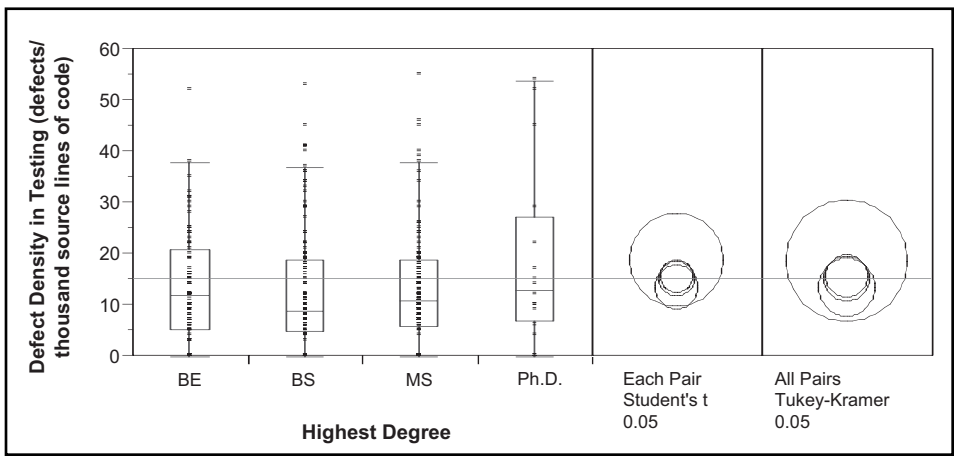
This result indicates nothing about whether people who pursue additional academic credentials will improve their ability; it simply indicates that they are not useful distinguishers between students in PSP. Although degree credentials are not a significant factor for PSP assignments, educational credentials may contribute to improved performance for more complex programs where deeper, more extensive domain or engineering knowledge may be crucial to understanding both the problem and potential solutions.

Years of Experience

Although some researchers have empirically found that *years of experience* are related to quality, it was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers. When the studies finding that experience was a significant factor were performed in the 1970s and 1980s, entry-level programmers were relatively unfamiliar with computers. The familiarity of the general population with computers has grown markedly over the last few decades. Students and entry-level programmers in the 1990s were likely to be well acquainted with computers before beginning their professional careers. The consequence is that the operational definition of *years of experience* for programmers is likely to have shifted in the last three decades. The results of the earlier studies may have been valid and yet be irrelevant to today's population of programmers.

This does not imply that experience does not affect ability. Holmes found that for his PSP data, collected over a seven-year period, developer experience was a significant factor [5]. His results, however, apply to a single individual engaged in applying PSP as part of a systematic improvement program. They indicate that individual professionals can substantially improve their performance over time, but they cannot be generalized

Figure 3: *Differences for Highest Degree Attained*



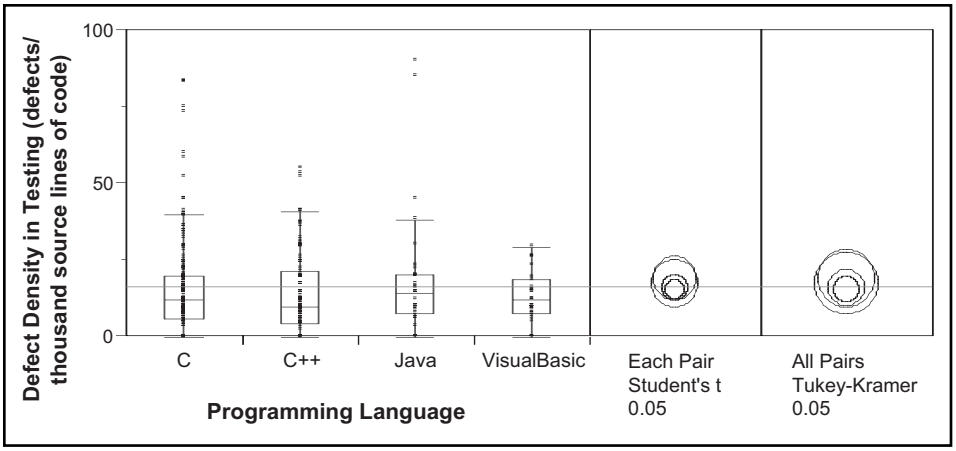


Figure 4: *Differences Between Programming Language Used*

across different individuals, which is the point of this analysis.

Industry studies of the effect of *years of experience* on quality typically use the average number of years for the team [6]. In averaging experience across the team, related factors in assigning professionals to the team with diverse backgrounds may impact the operational meaning of *experience*. The relevant factors may be related to a diverse team with a variety of skills and capabilities – *years of experience* may be confounded with other factors related to the skills of the team.

Number of Languages Known

The *number of languages known* may suggest the diversity of experience of the programmer, which in turn may indicate ability. While length of experience is usually considered a poor surrogate for ability, breadth of experience is considered a more realistic indicator [7], particularly for programmers with less than three years of experience [8]. The number of languages that a programmer has a working knowledge of may be considered a reasonable surrogate for breadth of experience. The *number of languages known* was shown to have a statistically significant effect on defect density in testing in only one of the four data sets analyzed, including outliers, and in none of the data sets excluding outliers.

Percent of Time Programming in the Previous Year

Recent experience in programming is primarily a concern for learning curve effects associated with programming skills in general. Given the small size of the PSP assignments, it seems likely that the bulk of any learning curve effects associated with basic programming skills are concentrated in the first few assignments. *Percent of time programming in the previous year* was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers.

Programming Language

Although some researchers have noted that the *programming language* used does not appear to be significantly correlated with productivity or quality except at a gross level [9], others have found that programming language can have a significant impact on both. For example, in defining the backfiring technique for estimating function points based on LOC, Jones found noticeable differences between different languages: It takes 128 LOC in C to implement one function point, but only 53 in C++ [10]. As illustrated by the overlap in the comparison circles in Figure 4, the *programming language* used was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers.

Although language may not significantly affect defect density, the productivity difference between languages implies that the number of defects will vary significantly depending on the language(s) used. In other words, if it takes twice as many LOC to implement a program in language A as in language B, then there will be twice as many opportunities for defects in the language A program as for the language B program, even if the defect density is the same. Because of the impact of programming language on productivity, and the related effects on process variables such as review rates, the *programming language* used should not be ignored in analyzing software quality, although it may not be a statistically significant variable for quality (as measured by *defect density in testing*) when considered in isolation.

Conclusion

This research found that (1) process discipline is an important factor for software quality; (2) *program size*, which is an indicator of solution complexity, is an important quality factor; and (3) *programmer ability* is an important quality factor when empirically measured. Other variables that may appear to be plausible surrogates for important areas such as ability and technology were not shown to be significant.

The issue of relative ability of programmers is particularly important, since the finding that even top-quartile performers improve more than 2X refutes those who resist the need for discipline, while acknowledging that their performance is superior and their opportunities for improvement are less than many of their colleagues.

This research supports the premise of PSP and similar process improvement strategies: Disciplined software processes result in superior performance compared to *ad hoc* processes. This improvement can be seen in both improved performance and decreased variation. It can be inferred that this is the minimum level of improvement that can be expected for a set of programmers since continual improvement can be expected after the PSP class.

The practical implications of this research for software managers and professionals are relatively simple, although they may be challenging to address. First, although *programmer ability* is a crucial factor affecting software quality, surrogates such as seniority and academic credentials are inadequate for ranking programmers, and empirical measures that are more effective may cause dysfunctional behavior if used for determining raises and promotions [11]. Second, consistent performance of recommended engineering practices improves software quality, even for top performers, who may resist discipline and measurement-based decisions because of their superior performance.◆

References

1. Paulk, M.C. "An Empirical Study of Process Discipline and Software Quality." Ph.D. diss., University of Pittsburgh, 2005 <http://etd.library.pitt.edu/ETD/available/etd-07082004-155917>.
2. Humphrey, W.S. A Discipline for Software Engineering. Reading, MA: Addison-Wesley, 1995.
3. Boehm, B., C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. Software Cost Estimation With COCOMO II. Upper Saddle River,

NJ: Prentice Hall, 2000.
4. Hayes, W. and J.W. Over. "The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers." Pittsburgh, PA: Software Engineering Institute, 1997.
5. Holmes, J.S. "Optimizing the Software Life Cycle." ASQ Software Quality Professional 5.4 (Sept. 2003): 14-23.
6. Zhang, X. "Software Reliability and Cost Models with Environmental Factors." Ph.D. diss., Rutgers, 1999.
7. Curtis, B. "The Impact of Individual Differences in Programmers." Working With Computers: Theory Versus Outcome. Ed. G.C. van der Veer. London: Academic Press, 1988: 279-294.
8. Sheppard, S.B., B. Curtis, P. Milliman, and T. Love. "Modern Coding Practices and Programmer Performance." IEEE Computer 12.12 (Dec. 1979): 41-49.
9. DeMarco, T., and T. Lister. Peopleware. 2nd ed. New York: Dorset House, 1999.
10. Jones, C. "Backfiring or Converting Lines of Code Metrics Into Function Points." Burlington, MA: Software Productivity Research, Oct. 1995.
11. Austin, R.D. Measuring and Managing Performance in Organizations. New York: Dorset House Publishing, 1996.

About the Author

Mark C. Paulk, Ph.D., is a senior systems scientist at the Information Technology Services Qualification Center at Carnegie Mellon University. From 1987 to 2002, Paulk was with the Software Engineering Institute at Carnegie Mellon, where he led the work on the Capability Maturity Model® for Software. Prior to joining Carnegie Mellon, he was a senior systems analyst for System Development Corporation at the Ballistic Missile Defense Advanced Research Center in Huntsville, Ala. He is a Senior Member of the Institute of Electrical and Electronics Engineers and a Senior Member of the American Society for Quality. Paulk has a Bachelor of Science in mathematics and computer science from the University of Alabama in Huntsville, a Master of Science in computer science from Vanderbilt University, and a doctorate in industrial engineering from the University of Pittsburgh.

IT Services Qualification Center
Carnegie Mellon University
Pittsburgh, PA 15213
Phone: (412) 268-5176
E-mail: mcp@cs.cmu.edu

CALL FOR ARTICLES

Do You Know How to Merge Hardware and Software?

CROSSTALK is looking for an introductory article that discusses merging hardware and software. Since most of CROSSTALK's articles come from our readers, we know this is our best way to reach you with this request. Call CROSSTALK Associate Publisher Beth Starrett at (801) 775-4158 or e-mail <beth.starrett@hill.af.mil> to discuss your experience in merging hardware and software, and how to transform your knowledge into an article that can get your ideas out to more than 100,000 CROSSTALK readers.

Also, take a look at CROSSTALK's new Theme Calendar format at <www.stsc.hill.af.mil/crosstalk/theme.html>. We now provide a link to each monthly theme, giving greater detail on the types of articles we're looking for.