

# Using a Personal Software Process<sup>SM</sup> to Improve Performance

Will Hayes

Software Engineering Institute  
Carnegie Mellon University

**Abstract:** The use of software measurement and process definition by individual engineers is embodied in the Personal Software Process (PSP)<sup>SM</sup>, a collection of techniques and guidelines for individual software engineers to use in building software. This paper presents a brief overview of the PSP and summarizes data collected by engineers in order to illustrate the efficacy of the PSP. Implications for the use of methods associated with statistical process control are discussed, and the power of rigorous data collection by individual software engineers is highlighted through discussion of empirical data.

---

<sup>SM</sup> PSP and Personal Software Process are service marks of Carnegie Mellon University.

## 1. Overview of the PSP

The PSP, developed by Watts Humphrey [1] provides an operationalization of sound engineering principles that is accessible for the individual software engineer. This disciplined application of engineering principles includes a strong focus on measuring individual performance on a set of clearly delineated activities. Engineers follow their own defined processes, performing a series of analyses, using their own data to uncover opportunities to enhance their performance. This approach provides engineers with an individual improvement plan that is tied to their performance.

The PSP is learned and practiced in a 100-150 hour training course that introduces practicing engineers to personal process measurement and improvement through a series of process levels. These levels represent a gradual elaboration of process definitions, and additions of techniques designed to be used by individual engineers. The course contains ten programming assignments that allow the engineer to gain experience with these techniques.

Analysis reports submitted between each process level allow the engineer to study their performance and the changes that occur as they focus on the different aspects of their personal software process. As historical data accumulate, the size of the programs assigned, the time required to perform each activity as well as the defects introduced and removed during each phase are estimated at the start of each programming assignment.

The following four sections provide a summary of each process level from the training course.

### 1.1 Baseline Performance

In the first level, engineers are asked to employ their usual software development practices, while tracking detailed measurements on their performance in completing three programming assignments. The data collected from these assignments provide a baseline for each engineer to work with as they add and modify steps in their personal process. While some measures display a surprising similarity between engineers, the initial data collected during these three assignments show a great deal of variability in individual performance. This observed variation in individual performance adds credence to an individual level focus for software measurement.

For the majority of engineers, size and effort estimates tend to be optimistic. In the absence of statistically sound ways of using historical data, engineers tend to guess based on their successful experiences - focusing on how well they can solve technical challenges, rather than anticipating the pitfalls that lurk beneath the requirements. While productivity (measured in lines of code per hour) varies widely from engineer to engineer, the in-process defect density is much more consistent across engineers - with a median of one defect for every 10 lines of code. [2]

## 1.2 Personal Project Management

In the second level, engineers begin to experiment with linear regression as a tool for estimating software size and engineering effort. The estimation technique (PROxy-Based Estimation, or PROBE) is introduced along with a host of measures designed to provide detailed monitoring of the performance of the process being followed. Using a tailored version of Crosby's "Cost Of Quality" approach [3], engineers examine where they invest their time, as well as where software defects are introduced and removed.

While many engineers struggle with accuracy in their estimates for size and effort, estimation accuracy indicators show a more balanced occurrence of underestimates and overestimates during this part of the course. [2] The detailed understanding of where time is spent as well as the realization that their estimates tend to be optimistic (when based on intuitive use of past experience), leads engineers to approach the problem of estimation more objectively. It is during this point in the training that engineers also discover the impact of product quality on the predictability of their work.

## 1.3 Personal Quality Management

The third level is used to implement and analyze techniques for managing product quality. Data and lessons learned from the previous six programming assignments are used to create design and code review checklists for use in completing the three assignments at this level. Based on the insights provided by their own data, engineers find efficient methods to prevent the time-consuming defects they had previously experienced at the end of the lifecycle. An explicit focus on design techniques and design verification leads to more complete designs that are verified prior to coding, and the impact of this investment is measured by each engineer as they produce high quality products in a more predictable fashion.

Typical engineers remove the vast majority of the defects in their products during the compile and unit test phases at the start of training. By the end of the training course, engineers are typically removing 30% to 75% of existing defects prior to these phases. [2] The data gathered from the first six assignments, and the defect analyses conducted, serve to illustrate the impact of poor product quality on the schedule the engineers set for their development work. The consequence of the rework during compile and test on the productivity and predictability of engineers' work are the chief motivators that lead engineers to strive for 100% pre-compile yield (removal of all existing defects prior to the compile phase).

## 1.4 Application to Larger Projects

The tenth and final assignment provides an opportunity to apply the techniques learned during training on a larger more complex program, using an iterative development cycle. The engineers use this final programming assignment to gain experience with ways to scale their personal processes to meet the needs of their work environment.

One of the chief lessons learned during this final exercise is an understanding of how to decompose the work tasks into manageable units. The engineers study their performance from the training course to define boundaries of optimal task duration and product size. This examination of their own planning and tracking, together with the focus on quality and project management gained from earlier lessons in the course, allows the individual engineer to approach their work assignments with superior knowledge of their own capability.

## 2. Effectiveness of PSP Training

The effectiveness of PSP training for changing the predictability and quality of the work of individual software engineers was evaluated using a sample of 298 engineers [2]. Data gathered by engineers as they completed the PSP training were used to evaluate changes in 5 areas of primary interest:

- Size estimation accuracy
- Effort estimation accuracy
- Product quality
- Early defect removal
- Productivity

The sample used in this study was a convenience sample, and did not represent a random selection of all software engineers. All engineers included in the study were trained by SEI personnel, or instructors who were authorized to deliver the course by the SEI. Eleven of the 23 classes in the data set were conducted as undergraduate or graduate offerings in major universities. Of the remaining 12 classes, 4 were instructor training classes offered at the SEI, and 8 were delivered to practicing engineers.

The lifecycle phases included in the PSP training are limited to planning (including hi-level design), detailed design, design review, coding, code review, compile, unit test, and postmortem. Requirements for the programming assignments are provided in the text book, and integration & system testing are never performed during the course. The focus in the PSP is on the phases in which an individual software engineer typically performs the bulk of his/her development work. Therefore, the discussion of class data provided here is limited to the phases listed above. The dis-

cussion of field data, later in the paper, addresses integration testing and post-deployment in addition to the phases listed above.

## 2.1 Size Estimation Accuracy

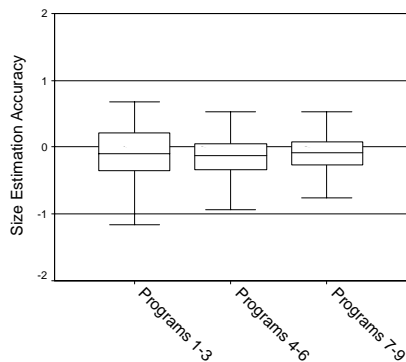
Estimating the size of a product prior to estimating the amount of effort required to build it is a relatively elementary notion. However, in the software industry the challenge of quantifying the size of the product being developed is sometimes abandoned in favor of an exclusive focus on the amount of effort required to build it. In PSP training, engineers learn how to use their personal data to estimate the size of products they build. The measure of product size used in the PSP is lines of code (LOC).

As described in section 1.1, estimates at the start of the training tend to be optimistic, with the majority of engineers underestimating the size of the software components they build. The figure below shows the data for size estimation accuracy, which was computed as:

$$\text{Actual LOC} - \text{Estimated LOC} / \text{Estimated LOC}$$

In the analyses reported here, the values of estimated LOC and actual LOC were summed across the three assignments in each PSP level, and the measure of estimation accuracy was computed using these pooled data. (All analyses of class data presented here employ this pooling method in order to characterize changes across the PSP levels rather than the variation across individual programming assignments)

**Figure 1: Size Estimation Accuracy Trend**



As the boxplots above show, the variation in size estimation accuracy for the first three assignments tends to be wide. The location of the median for the first set of assignments (the horizontal line through the first box) shows that the majority of engineers underestimated the size of the programs.

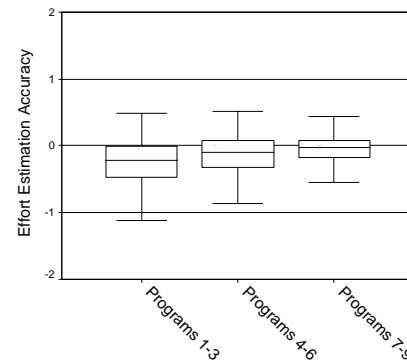
An analysis of variance, with repeated measures, found statistically significant differences in size estimation accuracy across the three groups of assignments ( $p=.041$ ). Post-

hoc analyses found that the difference between the first and second group of assignments (assignments 1-3 vs. assignments 4-6) is statistically significant as well ( $p=.037$ ). Post-hoc comparison of the second and third group of assignments revealed no statistically significant differences in size estimation accuracy. The ANOVA was replicated using transformed values of size estimation accuracy (to provide a more nearly-normal distribution), and these results are consistent with the original findings.

## 2.2 Effort Estimation Accuracy

Much like the pattern observed in size estimation accuracy, the effort estimates of engineers early in the course tend to be optimistic, and as the figure below illustrates, this bias toward underestimation changes to a more balanced occurrence of over- and underestimates by the end of the training.

**Figure 2: Effort Estimation Accuracy Trend**



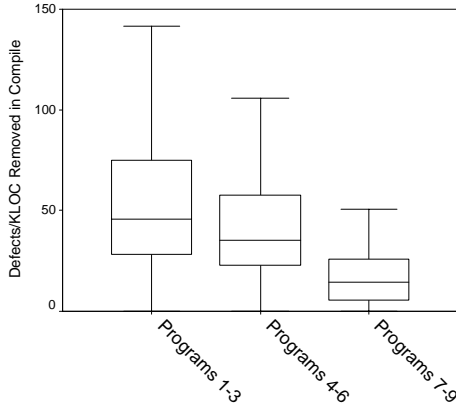
The results of the ANOVA conducted for these data are consistent with the results of the analysis of size estimation accuracy. Statistically significant differences across the three groups of assignments were detected ( $p<.0005$ ), and the post-hoc comparison between the first group and the second group was statistically significant as well ( $p<.0005$ ). Reanalysis of transformed data also confirmed these results.

## 2.3 Product Quality

The quality of a product throughout its development has a substantial impact on the cost of development. During PSP training engineers record and analyze data for each defect they discover in the programs they write. A careful focus on the effectiveness of defect removal activities guides the definition of personal processes. One explicit goal in the training is to reduce the reliance on the compile and test phases for removal of defects. Ideally, each engineer strives to remove all defects prior to compiling the program for the first time. This early defect removal trend

is addressed in the next section. Changes in the number of defects per thousand lines of code (KLOC) removed during the compile and unit test phases are addressed here.

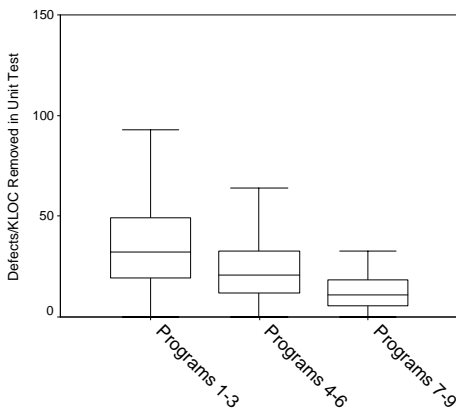
**Figure 3: Defects/KLOC Removed in Compile**



As is clearly obvious in the boxplots above, there is a reduction in the reliance on the compile phase for defect removal as engineers progress through the training. The ANOVA carried out for these data found statistically significant differences across the three groups of assignments ( $p < .0005$ ). Post-hoc comparisons of adjacent groups of assignments also found statistically significant differences ( $p < .0005$  for both).

The analysis of defects per thousand lines of code removed during the unit test phase revealed the same results. Re-analysis of the transformed data also confirm both sets of results.

**Figure 4: Defects/KLOC Removed in Unit Test**



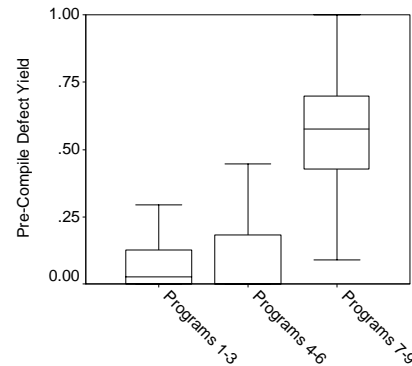
The boxplots shown in Figures 3 and 4 illustrate substantial reductions in rework during the compile and unit test phases. The median reduction for the compile phase was an improvement factor of 3.7. This summary statistic is the median of the distribution of ratios computed for each engineer. The ratios were computed by dividing the defects/KLOC removed in compile for the first group of

assignments, by the defects/KLOC removed in compile for the last group of assignments. The comparable summary statistic for the unit test phase is a median reduction of 2.5.

## 2.4 Early Defect Removal

Published data on the relative cost to remove a defect, depending on the phase in which it is detected, suggests that strategies to remove defects early are more cost-effective.[4] The findings presented in the previous section illustrate the level of reduction achieved in 'late defect removal.' In this section, analyses of yield provide a means of characterizing the changes in early defect removal effectiveness. Yield is defined as the percentage of defects that are known to have existed prior to the compile phase, that were removed prior to the first time the engineer compiles his/her code.

**Figure 5: Trend in Yield (early defect removal)**



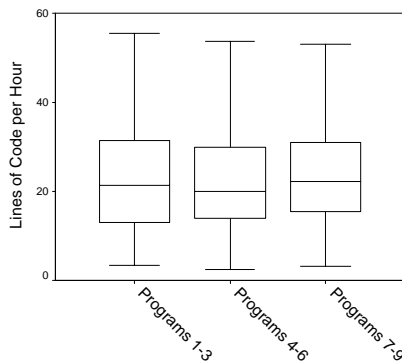
The boxplots above illustrate the heavy reliance on the compile and unit test phases for defect removal at the start of the training course. The values of yield for the first two groups of assignments are typically below 25%, and in fact the median for the second group of assignments is zero. As Figure 5 illustrates, the yields observed for the third group of assignments are generally higher for the group. The majority of engineers remove half or more of the defects prior to compile during the final group of assignments (The median value for the third distribution is approximately 58%). The ANOVA conducted with these data found statistically significant differences across the three groups ( $p < .0005$ ), and the post-hoc comparison between the second and third groups of assignments was also significant ( $p < .0005$ ). In order to guard against spurious results associated with failure to meet the statistical assumptions of the ANOVA, the Page test for ordered alternatives (a non-parametric alternative to repeated measures ANOVA) was used to confirm these results.[5]

## 2.5 Productivity

The final area of focus for the study of training effects is productivity. Despite the valid concerns for the sufficiency of measures used to represent productivity in the software industry, the use of lines of code per hour (LOC/Hour) by individual engineers as a measure to guide planning and monitoring of their personal work is beneficial. In the limited scope of an individual engineer, characterizing the rate at which s/he works can be useful. The analysis of productivity based on class data are difficult to generalize beyond the classroom setting, as the assignments completed for the course did not undergo integration, system, or acceptance testing. Data collected from field use of the PSP demonstrates that reduction in effort required in these critical phases is the true productivity pay-off associated with high quality software components. [6]

The analysis of productivity is presented to examine the possibility of a trade-off between quality and efficiency. With the addition of personal design reviews and code reviews, the concern for reduced productivity is raised. The figure below presents the trend in productivity across the three groups of assignments.

**Figure 6: Productivity Trend**



The boxplots in the figure above do not suggest any notable change in the distribution of productivity across the three groups of assignments. The repeated measures ANOVA supports this observation. The analysis found no statistically significant difference across the three groups of assignments.

## 2.6 Summary of Training Analysis

The analyses described above provide evidence that the performance of individual engineers changes during PSP training. Furthermore, these changes are consistent with the objectives of the PSP training course. Substantial gains in predictability and quality are observed, with no reduction in productivity.

These findings are compelling for instructors and students of the PSP training class. However, software development organizations require evidence that similar improvements can be achieved in the daily work of these engineers. The remainder of this paper is focused on the evidence of this real-world impact.

## 3. Effectiveness of PSP in the Field

The case study described here is based on detailed measures collected by two PSP-trained engineers working to build 26 components for a corporate information system. The combined effort of these two engineers represents 877 direct project hours, creating 10,390 lines of new C code. The lifecycle model used in this project resembles the traditional waterfall model. Each of the 26 components was built in sequence, then integrated into a complete system at the end.

Because the product built by these engineers has been in use for over 18 months, the defect data collected by the engineers provide an opportunity to characterize the quality of the process used. During the development and integration of the 26 modules, 504 defects were discovered and removed. To date, no defects have been discovered in the field.

### 3.1 Predictability Performance

Each component in the project was estimated separately using the PROBE method, which is described in great detail in Humphrey's book *A Discipline for Software Engineering* [1]. The project described here was delivered on time and within budget. The total effort required to implement the 26 components was 877 hours, and the estimate was 876.1 hours, as shown in the right-most column of Table 1.

**Table 1: Predictability Indicators**

Measure	Engineers		Pooled Project Data
	A	B	
Estimated LOC	4212	5575	9787
Actual LOC	3844	6546	10390
Size Estimation Accuracy	8.74%	-17.42%	-6.16%
Estimated Hours	268.1	608.0	876.1
Actual Hours	209.9	667.1	877.0
Effort Estimation Accuracy	21.71%	-9.72%	-0.10%

Looking at the data in Table 1, the effort estimation accuracy for the project as a whole implies an underestimate of one-tenth of one percent. While this value implies an accurate estimate of some precision, examination of estimation accuracy for each engineer reveals that the estimation errors of the two engineers tend to balance each other. Engineer A overestimated both size and effort, while engineer B underestimated both size and effort.

Accurate estimates for size and effort are difficult to obtain. The engineers described here were able to gather a fairly large set of data and employed linear regression to estimate their work during the latter half of the project. The data collected and used by the engineer who built 19 of the components are used to illustrate the use of linear regression in estimating.

When using the PROBE method, the primary input to the estimation algorithms (besides the historical data base which is well described in [1]) is a projected LOC value. This value is derived from the high level design work done by the engineer. Each object in the design is characterized by object type (e.g., I/O versus Calculation objects) and by relative size (e.g., small, medium or large). Based on historical LOC counts for objects of the same type and relative size, a total projected LOC count is derived. This value is then used as the basis for all other estimates.

Based on the data collected from 19 components, the linear regression equation which expresses the relationship between projected LOC and actual LOC for this engineer is

$$\text{Actual LOC} = 67.4 + 1.1 (\text{Projected LOC})$$

The regression yielded a statistically significant multiple correlation of .85 ( $p < .00005$ ), which when squared suggests that some 72% of the variation in actual LOC can be accounted for by the variation in Projected LOC.

Looking at the relationship between Projected LOC and actual effort, this engineer's data suggest the following equation

$$\text{Actual Effort} = 8.8 + .1 (\text{Projected LOC})$$

While statistically significant, the multiple correlation coefficient of .69 ( $p = .002$ ), suggests that only 47% of the variation in actual effort can be explained by the variation in Projected LOC. The engineer would therefore not rely on this relationship to estimate effort, while the previous regression equation does provide a viable estimate of the actual size of the program to be written. The average productivity to date, expressed in lines of code per hour, may serve the engineer better in this case, because of the established relationship between actual size and actual effort. For this engineer, this relationship is expressed in the regression equation below

$$\text{Actual Effort} = -1.2 + .1 (\text{Actual LOC})$$

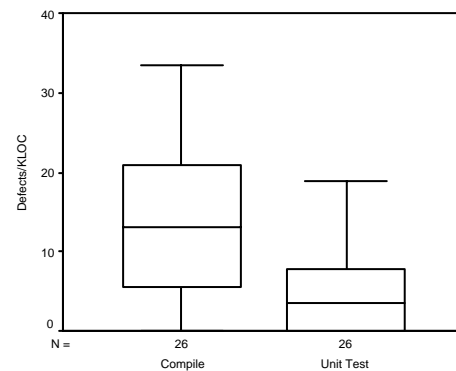
The multiple correlation coefficient for this equation equals .91, and it was found to be statistically significant ( $p < .00005$ ).

While statistically significant results from regression analysis, accompanied by high values of R-squared are important from a statistical perspective, operational use of these statistics requires more detail. Each computed multiple correlation has an error variance associated with it, and ignorance of this variation can lead to unexpected results. For this reason, each regression-based estimate is accompanied by a prediction interval. As a result, each engineer uses a point estimate and associated prediction limits (usually a 70% as well as a 90% prediction interval) to plan the size and effort required for each project they complete. Engineers who use the PSP report a dramatic improvement in the confidence they associate with their estimates. These methods are also contributing to changes in management expectations as well as the level of involvement enjoyed by project team members at the beginning of each project.

### 3.2 Product Quality

As indicated above, the product built by these engineers has no reported field defects to date. In this section, the number of defects per thousand lines of code (defects/KLOC) of the 26 components in the compile and unit test phases are summarized.

**Figure 7: Defect Density in Compile and Unit Test**



The boxplots above show the distributions of defects/KLOC removed during the compile and unit test phases. Comparing these data to the end-of-class performance summarized in section 2.3, we see that the performance of these engineers in the field is comparable (if not

superior). The table below compares the median defect densities for these two phases, for class data and the field data summarized here.

**Table 2: Median End-of-Class and Field Defect Densities in Compile and Unit Test**

	Course Data	Field Data
Defects/KLOC in Compile	14.56	12.99
Defects/KLOC in Unit Test	10.87	3.55

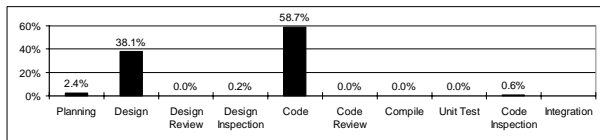
Of course, the data summarized above represents only the defects that have been discovered in the software components during the phases listed. The delivered product quality of the class assignments will never be known. The components built in the field are in use, and have proven to be of high quality.

### 3.3 Early Defect Removal

In this section, a detailed examination of defect introduction (injection) and removal is provided. The engineers who supplied these data use this type of analysis to understand the effectiveness of their personal processes, and to make modifications to their processes in order to improve their performance.

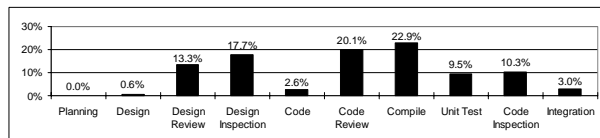
Figure 8 illustrates the pattern of defect injection, summarized by lifecycle phase. Data for all 26 components are included in this figure.

**Figure 8: Defect Injection Profile**



The vast majority of defects (97%) are introduced during the design and coding phases.

**Figure 9: Defect Removal Profile**



The distribution of defect removal activity across the lifecycle illustrates the quality of the process used to develop this product. The tendency to rely almost exclusively on the compile and unit test phases for defect removal, which we observe in many engineers at the start of PSP training, is not apparent in these data.

**Figure 10: Average Fix Time by Phase of Removal**

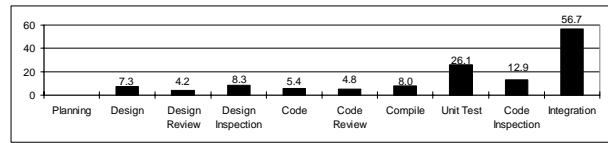


Figure 10 provides empirical support to an established axiom in software engineering; that defects removed late in the lifecycle are more costly than defects removed early in the lifecycle. The average time to remove a defect during integration was 56.7 minutes. Defects removed during design review, on the other hand, required an average of 4.2 minutes.

Finally, the data charted in Figures 8 and 9 support an important conclusion about the quality of the process being used by these two engineers. As implied by Figure 8, approximately 41% of all defects were injected prior to the coding phase. As implied by Figure 9, approximately 32% of all defects were removed prior to the coding phase. With only 9% of all defects “escaping” into the coding phase, this pattern is indicative of a process where the design work provides a fairly ‘clean’ input to implementation. This type of analysis of empirical data allows engineers to characterize the effectiveness of the process they use in quantitative terms.

To illustrate the effectiveness of each defect removal phase, Table 3 lists the percentage of existing defects that were removed during each ‘defect removal’ phase. The percentages in the table are calculated by dividing the number of defects removed in the phase by the number of defects that are known to have existed during that phase. The first two columns of percentages summarize the performance of each phase for the individual engineers. The third column summarizes the patterns for all 26 components by pooling the data for all components before com-

puting the percentage. (The columns for each engineer represent pooled data for only the components built by that engineer).

**Table 3: Defect Removal Percentages**

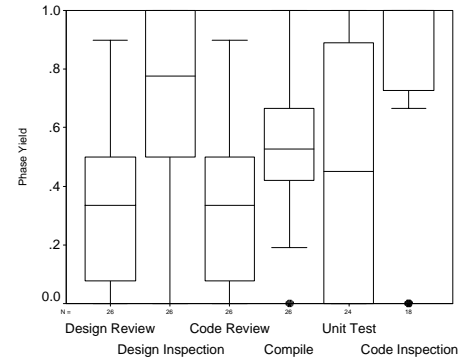
Lifecycle Phase	Percent of Defects Removed in each Phase		
	Engineer A (7 components)	Engineer B (19 components)	Project Totals
Design Review	44.9%	26.0%	33.3%
Design Inspection	60.5%	68.5%	65.9%
Code Review	38.2%	28.5%	30.7%
Compile	48.9%	50.8%	50.4%
Unit Test	25.0%	47.2%	42.5%
Code Inspection	83.3%	74.0%	76.5%

Relative to all other phases, the code inspection and design inspection phases are most effective in terms of the percentage of defects that are detected. Note that the design review and code review phases are both more effective for engineer A, relative to engineer B.

The percentages listed in Table 3 summarize the performance of each of the listed phases for the project as a whole. In fact, the data from each component built provides an opportunity to characterize the defect detection effectiveness of the phases for a given instance of its use. Understanding the variability in the effectiveness (as well as the typical level of effectiveness) has important implications for planning and tracking software project performance.

In the figure below, boxplots are used to show the distribution of yield for each of the 6 phases studied here. The number of observations contained in each plot (shown directly below the horizontal axis) varies due to the fact that no defects were present in some of the components as they entered some of the phases. For example, the distribution of code inspection yield is based on only 18 components because no defects were present in the code for 8 of the components as they entered that phase.

**Figure 11: Distributions of Phase Yields**



While the percentages reported in Table 3 suggest that the unit test phase was more effective than the review phases overall, the distribution of yield for the unit test phase in the figure above suggests that there is a great deal of variability in effectiveness. In contrast, the effectiveness of the two inspection phases (design inspection and code inspection) are consistently high.

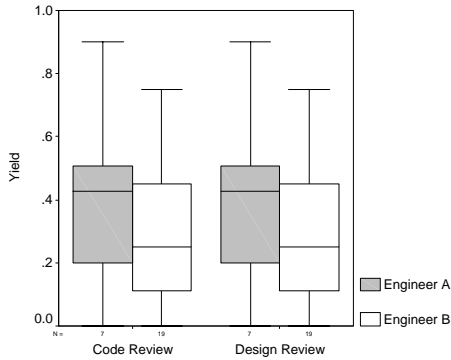
The inspections conducted for this project employed an established methodology in the organization. The two engineers who built this product had substantial training and experience with this methodology, and conducted the inspections together. It is not surprising that these phases tend to consistently perform well.

The reviews (Design Review and Code Review) were learned as a part of the PSP training. These reviews are personal reviews conducted by a single engineer, based on their own historical defect data. The distributions for these individual review phases are nearly identical (in Figure 11). As noted in the discussion of Table 3, however, the reviews conducted by engineer A seem to be more effective in identifying defects.

This understanding of the performance of defect-removal processes allows the engineers to make changes to their personal processes in order to improve their performance. Changes to the unit test process may be targeted to reduce the variability in defect detection performance, and changes to the personal review processes may be targeted to improve their general level of effectiveness.

A more detailed comparison of the effectiveness of the personal review processes of these two engineers is shown below.

**Figure 12: Differences in Yield Distributions**

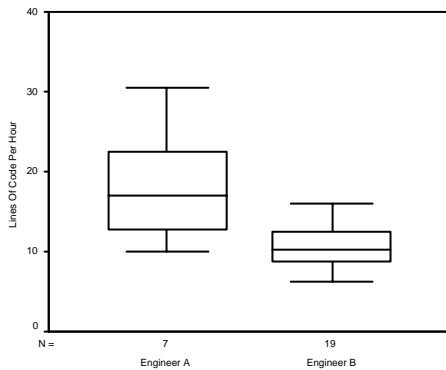


At the end of this project, Engineer B worked with a process coach in the organization to improve the performance of his design reviews and code reviews. Using the data summarized in this paper, this engineer was able to find ways to increase the yields for both of these phases. On the 9 components built by engineer B following this project, the median yield of the design review process was 57%, and the median yield of the code review process was 54%. These values are more similar to the performance of the review processes used by engineer A, shown in the figure above.

### 3.4 Productivity

The distribution of LOC/Hour for each engineer are shown in the figure below.

**Figure 13: Productivity Distributions**



In general, engineer A enjoyed a higher LOC/Hour rate than engineer B did. However, the variability in the productivity of engineer A is substantially greater. These observations may reflect differences in personal work style as well as differences in the components being built by the two engineers.

Regression analyses using the percentage of defects removed in each phase, as well as the defect detection capability of the review phases to predict the productivity rate yielded no statistically significant results.

### 3.5 Summary of Case Study

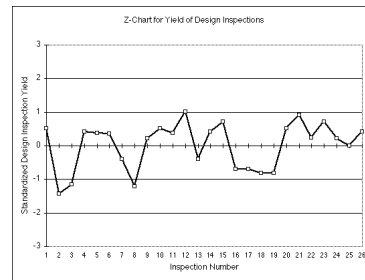
The process definition and measurement techniques learned during PSP training serve these engineers well in their work. Detailed data collected during the course of the project were used to monitor and improve their performance. Understanding the typical performance of process steps like reviews and testing, as well as the variability in their performance arms the engineer with superior knowledge of their capability.

## 4. Application of Statistical Process Control

The utility of statistical process control techniques for performance monitoring and improvement by individual engineers is explored in this final section. The data collected from the design inspections performed by these engineers provides an opportunity to illustrate how process stability may be assessed, and the impact of process changes may be anticipated. The fundamental premise behind this discussion is that systematic changes to a stable process should result in predictable changes in performance.

As noted in the discussion of Table 3, the inspection processes used by these engineers performed very well with respect to their ability to discover defects. Figure 14 charts this performance across the 26 components using a control chart. A Z-chart was selected because it accommodates observed values arising from a process where the area of opportunity varies from sample to sample. A U-chart also has this attribute, and might be employed for the same purpose. For more information on the construction and interpretation of control charts, refer to [7].

**Figure 14: Trend in Design Inspection Yield**



The above chart depicts the trend in the yield of design inspections. The data imply a very stable process, with few deviations beyond the 1-sigma limits. (The value of the mean for these data is 65.9%).

Tables 4 and 5 provide a summary of defect data by defect type. Table 4 shows the number of defects escaping design inspection, the number of defects discovered in design inspection, and the percentage of each type of defect that was captured. Table 5 shows the average rework effort

required to fix defects, by defect type. These tables contain data for only those defects that existed in the products at the time of the design inspection.

**Table 4: Escapes and Captures for Design Inspections**

Defect Type	Escaped	Captured	Yield
Documentation	6	8	57%
Syntax	4	23	85%
Assignment	4	5	56%
Interface	2	6	75%
Checking	0	4	100%
Data	1	1	50%
Function	19	38	67%
Standards	6	11	65%
Consistency	1	1	50%

**Table 5: Average Rework by Defect Type (in minutes)**

Defect Type	Escaped	Captured
Documentation	3.33	1.38
Syntax	5.00	5.13
Assignment	15.75	5.60
Interface	10.00	4.67
Checking	N/A	12.50
Data	10.00	25.00
Function	17.89	13.29
Standards	8.17	3.82
Consistency	5.00	2.00

Examining the data in Table 4, we see that the largest number of defects escaping the design inspection are “Function” defects. Furthermore, when we look at Table 5, we see that these defects are associated with the largest average fix time when they escape the design inspection process - requiring an average of 17.89 minutes to repair if they escape, and an average of 13.29 minutes if they are discovered during design inspection. These two observations suggest that process improvements aimed at preventing these defects from entering the design inspection, or

more effectively capturing them during design inspection, are likely to result in observable performance improvements.

Given that the yield of the design inspection process appears to be stable (when we look at the control chart in Figure 14), we might wish to estimate the likely performance gain associated with reducing the impact of “Function” defects. The engineers would search for ways to reduce the occurrence of these defects, or ways to discover a higher percentage of them during design inspections. Because the process seems to be stable, improvements to the process may result in predictable changes in the empirical data collected by the engineers.

If changes can be made to the inspection process, to more effectively discover “Function” defects, the resulting performance gain will depend on the distribution of these defects across the components. That is, if every component contains one or more “Function” defects, an improved inspection process is more likely to have a homogeneous improvement in design inspection performance. If, on the other hand, these defects occur in only a subset of components, then an improved design inspection process is more likely to show a heterogeneous improvement. Referring to the control chart in Figure 14, a homogeneous improvement in design inspection performance would manifest itself in an increased average yield for all design inspections. A heterogeneous improvement in design inspection performance would manifest itself in a dampening of the variation around the center line in Figure 14.

On the other hand, if efforts to prevent “Function” defects are successful, we might expect no change in the performance of design inspections, if the presence of these defects is homogeneous across all components. Conversely, if these defects are present in only a subset of components, we might expect that results of preventing these defects would tend to reduce the variability in the performance of design inspection.

All of the above projections are predicated on the stability of the design and design inspection process over time. In this context, the term process is used to encompass more than the series of procedures and methods applied to create and inspect a design. Stability in the influence of contextual factors such as engineers’ abilities, and attributes of the product would also be required - at least to the extent that these factors have an influence on the performance of design inspections (and indeed design creation). With the rigorous collection of measurements, as described in this case study, these important issues can be examined empirically by each engineer.

## 5. Summary

In conclusion, this paper shows that individual engineers who employ a rigorous measurement framework to monitor their performance can create opportunities to substantially enhance their performance.

Implications for the application of statistical process control were illustrated to discover process improvement opportunities, and to estimate their likely impact. The use of linear regression was illustrated using the historical data collected by an engineer, who was himself using regression to derive estimates.

As the data collection and analysis associated with the PSP continues, we expect to broaden the focus of the work described in this paper. In addition, more detailed insights will be made available through the work of individual engineers who take ideas like these and apply them to their professional work.

## References

- [1] Watts S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley, Reading, MA, 1995
- [2] Will Hayes and James W. Over, *The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers*. CMU/SEI-97-TR-001 (available via <http://www.sei.cmu.edu/>)
- [3] Phillip B Crosby, *Quality Is Free, The Art of Making Quality Certain* (New York: Mentor, new American Library, 1979)
- [4] Moller, K.H. and Paulish, D.J., *Software Metrics: A Practitioner's Guide To Improved Product Development*, IEEE Press, 1993
- [5] Page, E.B. *Ordered Hypotheses for Multiple Treatments: A Significance Test for Linear Ranks.* *Journal of the American Statistical Association* 58, 301 (March 1963):216-230.
- [6] Furguson, Pat; Humphrey, Watts S.; Khajenoori, Soheil; Macke, Susan; & Matvya, Annette. *Introducing the Personal Software Process: Three Industry Case Studies*. *IEEE Computer* 30, 5 (May 1997): 24-31.
- [7] William A. Florac, Robert E. Park, and Anita D. Carleton, *Practical Software Measurement: Measuring for Process Management and Improvement*. CMU/SEI -97-HB-003 (available at <http://www.sei.cmu.edu/>)