Investigating Data Quality Problems in the PSP

Anne M. Disney
Dept. of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822 USA
+1 808-956-6920
anne@ics.hawaii.edu

Philip M. Johnson
Dept. of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822 USA
+1 808-956-3489
johnson@hawaii.edu

Abstract

The Personal Software Process (PSP) is used by software engineers to gather and analyze data about their work, and to produce empirically based evidence for the improvement of planning and quality in future projects. Published studies have suggested that adopting the PSP results in improved size and time estimation and in reduced numbers of defects found in the compile and test phases of development. However, personal experience using PSP in both industrial and academic settings caused us to question the quality of PSP data when manually collected and analyzed. To investigate this issue, we built a tool to automate the PSP and then examined 89 projects completed by ten subjects using the PSP manually in an educational setting. We discovered 1539 primary errors and categorized them by type, subtype, severity, and age. To examine the collection problem we looked at the 90 errors that represented impossible combinations of data and at other less concrete anomalies in Time Recording Logs and Defect Recording Logs. To examine the analysis problem we developed a rule set, corrected the errors as far as possible, and compared the original and corrected data. This resulted in significant differences for measures such as yield and the cost-performance ratio. Our results raise questions about the accuracy of manually collected and analyzed PSP data and point to directions for future research.

Keywords

Personal software process, errors, defects, empirical software engineering, measurement dysfunction, automated tool support

1 INTRODUCTION

> *The actual process is what you do, with all its omissions, mistakes, and oversights. The official process is what the book says you are supposed to do.* [5]

The Personal Software Process (PSP) was introduced in 1995 in the book, "A Discipline for Software Engineering" [5]. This text describes a one-semester curriculum for advanced undergraduates or graduate students in computer science that teaches concepts in empirically-guided software process improvement. Since its introduction, experience with the PSP has been reported on in several case studies [1, 3, 6, 9, 7]. Although empirically-guided software process improvement is a key feature of other software engineering initiatives, such as the Capability Maturity Model (CMM) [8], ISO-9000, and Inspection [4], the PSP differs from these other approaches in important ways.

The CMM, ISO-9000, and Inspection discuss empirical software process improvement in the context of a large organization. Process improvement in this context requires the gathering and analysis of large amounts of data, within and across departments, generated by different people at different times. Indeed, inevitable personnel turnover means that the data collected from the working procedures of one set of people tend to generate measurements leading to process changes that affect the working procedures of a potentially different set of people. The substantial effort required to collect, interpret, and introduce organizational change based upon the measurements for a large organization leads to the need for an explicit software engineering process group (SEPG) whose mission is to manage empirically guided improvement. Although the utility of these approaches have been repeatedly validated, they leave the unfortunate impression that empirically-guided software process improvement is the sole province of large organizations who can dedicate teams of people to its enactment.

The PSP provides an alternative, complementary approach in which empirically guided software process improvement is "scaled down" to the level of an individual developer. In the PSP, individuals gather measurements related to their own work products and the process by which they were developed, and use these measures to drive changes to their development behavior. PSP focuses on defect reduction and estimation accuracy improvement as the two primary goals of personal process improvement. Through individual collection and analysis of personal data, the PSP provides a novel example of how empirically-guided software process improvement can be implemented by individuals regardless of the surrounding organizational context and the availability of institutional infrastructure support.

Since PSP is a new technique, relatively little data exists on its use and effectiveness. Those studies of which we are aware all report positive results, usually based upon measurements obtained by application of the PSP process itself. For example, one case study states that "during the course, productivity improvements average around 20% and product quality, as measured by defects, generally improves by five times or more" [3]. Another study states that "the improvement in average defect levels for engineers who complete the course is 58.0% for total defects per KLOC and 71.9% for defects per KLOC found in test." Indeed, our own PSP data yields similarly positive measurements for process and products.

In this paper, we report on a case study performed to assess the quality of PSP data—the data often used in evaluations of the effectiveness of the PSP as shown above. Our case study was motivated by our experiences teaching and using the PSP, which led us to suspect that the empirical measures gathered by the PSP may not, in all cases, reflect the true underlying process or products of development. In our case study, we taught a modified version of the PSP curriculum augmented with mechanisms to ameliorate potential PSP data quality problems. We then entered the PSP measures into a database and subjected them to a variety of data quality analyses. These analyses uncovered over 1500 errors in the PSP data generated by the ten students in the class during nine projects. Additional analysis yielded a seven part classification scheme for PSP data errors. Although we were not always able to generate corrected values for the data errors, partial correction lead to substantially different values for certain PSP measurements. These results lead us to several conjectures about the nature of data quality in the PSP and implications for future development of empirically guided personal software process improvement.

The remainder of the paper is organized as follows. The next two sections present a brief overview of the PSP and a simple model we developed to organize our exploration of PSP data quality problems. The following three sections present the case study, its results, and our conclusions.

2 OVERVIEW OF THE PSP

In the PSP curriculum presented in "A Discipline for Software Engineering", each student develops 10 small programs over the course of a semester using a sequence of seven increasingly sophisticated software development processes. For every program, the students record various measurements related to their personal development activities. Such measures include, for example, the time spent in each phase of development, the numbers of defects injected and removed during each phase, and the size of the resulting work product.

The initial programs use relatively simple processes that establish a baseline set of process measures for time, size, and defects. Later programs employ more advanced processes that extend these baseline process statistics. Although there are a myriad of individual extensions, most fall into two conceptual categories.

First, the planning phase is expanded to include estimates of the program's projected size, the projected time required to complete each of the phases, and the number of anticipated defects that will be injected and removed during development. The process by which these estimates are produced involves statistical analysis of historical correlations between designs (i.e class and method counts) and actual size (in lines of code), between estimated size and actual time, between actual size and actual time, and between size and defects injected and removed. (While lines of code as a metric of size at the organizational level is almost uniformly excoriated in the measurement literature, it seems to work surprisingly well in the PSP, since the measure is collected and applied to a single individual working in a single language in a relatively uniform domain.)

Second, by the middle of the course, each student has typically recorded a hundred or more defects they have made during development. Later processes include mechanisms to help students understand the impact of various kinds of defects and to drive process improvements intended to reduce future occurrence of important defect types. For example, since students record the phase each defect was injected and removed and the time required to fix it, it is possible to analyze the relationship between fix time and various characteristics of defects. One relationship nearly always present in student data is that the "longer" a defect is present, the more time it takes to remove it. Thus, defects injected during design and not removed until testing are nearly always more expensive to remove than, for example, defects injected during coding and removed during compiling. This outcome typically motivates students to put more effort and care into design activities. Later processes support such behaviors by providing active defect management mechanisms. For example, by analyzing defect data to determine the types of design defects made on prior projects, a student can generate a checklist to be used as part of a personal design review to ensure that those defects do not escape into code, compile, or test phases.

The final stages of the course further extend the basic PSP paradigm. The last PSP process provides a way to scale the method to support larger projects using a cyclic development method. In addition, PSP includes a meta-level process for defining personal processes in non-software domains or for specific software organizational contexts.

3 A MODEL OF PSP DATA QUALITY

To guide our understanding of data quality problems in the PSP, we devised a simple two stage model of PSP data, as illustrated in Figure 1. The model begins with "Actual Work"—the actual developer efforts devoted to a software development project. As part of these efforts, the developer *collects* a set of primary measures on defects, time, and work product characteristics—the "Records of Work". Based on these primary measures, the developer performs additional *analyses*, many of which result in secondary (i.e. derived) measures which are themselves inputs for further analyses. The secondary, derived measures and associated analyses are presented in various PSP forms—the "Analyzed Work"—and hopefully yield "Insights about Work" to improve future software development activities.

We based this model upon the PSP as presented in *A Discipline for Software Engineering* [5] - what we term "manual PSP". Manual PSP refers to a situation in which the PSP forms must be filled out by hand, either by editing a copy of the form on-line, or by filling out out a printed copy with pen or pencil. Even if tools such as spreadsheets are used to collect historical data and to provide various computations, if they do not automatically insert and maintain the correct calculated values in the appropriate places in the forms, then we define the technique as "manual". We define partially or fully "automated" PSP as one in which some or all of the derived measures are calculated and placed into the forms automatically. In other words, in automated PSP, the analysis tools and forms presenting the PSP reports are tightly *integrated*. Although "automated" PSP can essentially automate all of the analysis stage calculations, there are limits to its ability to automate the collection stage work. The collection stage is still quite "manual" in nature.

At the time we performed this case study, there was no integrated software support for the PSP. Thus, the case study was based upon a "manual" version of the PSP, despite the presence of tools. Since then, integrated tools have become available, including spreadsheets available at the Addison-Wesley FTP site which print out the project summary forms, and the Personal Software Process Studio tool produced by East Tennessee State University.

There are three basic ways to affect PSP data quality in the collection stage: errors of omission, errors of addition, and errors of transcription. Errors of omission occur when the developer does not record a primary measure related to defects, time, or the work product itself. If a defect occurring during "Actual Work" does not appear in the "Records of Work", then, for example, the PSP model of that work product's defect density will be lower than its actual defect density. If time spent on the work product is not recorded, then the PSP model of that developer's productivity will be higher than her actual productivity. Errors of addition occur when the developer augments the "Records of Work" with data not reflecting

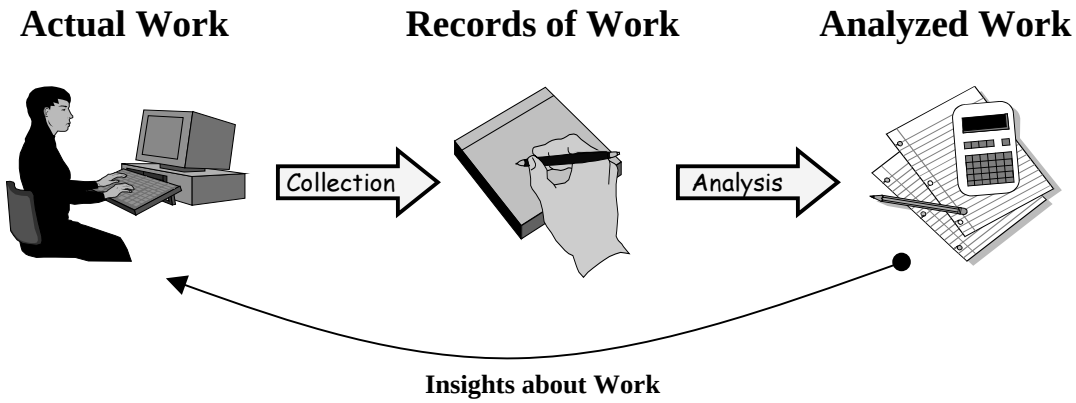

Figure 1: A simple model for PSP data quality. Through a process of *collection*, the developer generates an initial empirical representation ("Records of Work") of her personal process ("Actual Work"). Through additional *analyses*, the developer augments her initial empirical representation with derived data ("Analyzed Work") intended to enable process improvement through "Insights about Work".

actual practice. For example, a developer, having made an error of omission to the point of having no time or defect data, may recover by simply inventing enough time and defect entries to make his or her PSP data appear plausible. Finally, errors of transcription occur when the developer does intend to record their "Actual Work" in the "Records of Work" but makes a mistake during this process.

The presence of collection stage data quality problems is typically difficult to ascertain and difficult or impossible to rectify. In the PSP, primary data collection often feels both time consuming and psychologically disruptive. Many students complain that stopping to record defects disrupts their "flow" state, and that the time spent recording a defect—particularly for compilation stage errors—often exceeds the time spent correcting the defect. The PSP requires users to learn to constantly interleave "doing work" with "recording what work you are doing".

There are also three basic ways to affect PSP data quality in the analysis stage of manual PSP: errors of omission, errors of calculation, and errors of transcription. Errors of omission occur when the developer does not perform a required analysis of the primary data. Errors of calculation occur when the developer attempts to perform an analysis but does so incorrectly. For example, a developer might use a regression-based estimation method when the historical data is so uncorrelated that this method is invalid. Finally, errors of transcription occur when the developer makes a clerical error when moving data from one form to another.

Unlike the collection stage, analysis stage data quality problems are much easier to ascertain and correct, *provided that errors did not occur during the collection stage*. In other words, if one assumes that the work records accurately reflect the underlying work, then appropriate use of automated tools can go quite far to eliminating analysis errors of omission, calculation, and transcription. On the other hand, since the quality of these analyses are totally dependent upon the quality of the work records, overall PSP data quality could be quite low even if the analysis stage could be totally automated to eliminate all potential data quality errors.

4 CASE STUDY

To gain insight into the occurrence and significance of collection and analysis data quality problems, we conducted a case study. The case study began by teaching a modified version of the PSP designed to improve data quality. Next, we entered the data into a database that automated most analysis calculations and revealed the presence of a subset of the possible errors in student data. We then performed additional analyses on these errors to understand their cause and potential significance to PSP data values and the method itself.

4.1 The Modified PSP Curriculum

The projects used for this study were obtained from a software engineering class taught by Philip Johnson, in which the PSP was taught over the course of a semester using nine project assignments. There were ten students in the class, and 89 completed projects.

Because of the concern with data quality from prior experience teaching PSP, the instructor made four principal modifications to the standard PSP curriculum: (1) increased process repetition, (2) process annotations, (3) technical reviews, and (4) tool support.

Increased process repetition. In the standard PSP curriculum, students are assigned 10 programs during the semester (in addition to several midterm and final reports). Over the course of these ten programs, students practice seven different PSP processes, which meant that the development process used by the students changed for seven out of ten programs. In our initial experience with the PSP, we believed that the overhead of this almost constant "process acquisition" led to data errors and had a significant impact upon the overall data values. To ameliorate this problem, the modified curriculum included only five PSP processes, enabling students to practice most processes at least twice before moving on to a new one. The modified curriculum also included only nine programs instead of ten, providing additional time in each program for data collection and analysis.

Increased process description. In our initial experiences teaching the PSP, the instructor found that students had a great deal of trouble learning to do size and time estimation correctly. For example, PSP time estimation requires choosing between three alternative methods for estimation depending upon the types of correlations that exist in the historical process data from prior programs. To help resolve this and other problems, the instructor added four additional worksheets: (1) a Time Estimating Worksheet to provide a guide through the various methods of time estimation; (2) a Conceptual Design Worksheet to help in developing class names, method names, method parameters, and method return values; (3) an Object Size Category Worksheet to help in size estimation; and (4) a Size Estimating Template Appendix to provide a place to record planned and actual size for prior projects.

Technical reviews. At the completion of each project, students divided into pairs and carried out a technical review of each other's work. A two-page checklist facilitated this process. It included such questions as "Did the author follow the PSP Development Phases correctly?" and "Is the Projected LOC calculated

correctly." A second "Technical Review Defect Recording Log" form included columns for number, document, severity, location, and description. Students were given approximately 60 minutes to do the review. The technical review forms were submitted with the completed projects. The instructor reviewed the projects a second time for grading purposes, using the Technical Review Defect Recording Log to record any additional mistakes.

Tool support. Finally, the instructor provided four spreadsheets to support records of planned and actual data values. In addition, students were provided with well-tested tools to count lines of code for Java programs, to compare two versions of a Java program and report non-comment lines of code added and deleted, and to perform certain statistical analyses. (In the textbook PSP curriculum, students "bootstrap" their environment by implementing these tools themselves. While elegant pedagogically, this approach unfortunately introduces a potentially significant source of data quality problems, since these freshly developed tools with no usage history are used to generate many of the measures used in later data analysis.)

The instructor also emphasized data quality throughout the duration of the course. For example, he augmented the lecture notes in the Instructor's Guide with fully worked out examples of the PSP process data for a fictitious student to show how data is collected and analyzed for each assignment and accumulated over the course of the semester. He dedicated lectures to collection and analysis of data periodically throughout the semester. He regularly showed the class aggregate statistics on class performance. He met with students individually and in groups throughout the semester to go over their assignments and PSP data while they were in the midst of planning, design, code, compile, test, and/or postmortem; but prior to project turn-in. He uncovered and removed many, many PSP data errors through these meetings which are not counted in our results. He did technical reviews of every assignment's PSP data, and circulated problem reports throughout the semester summarizing issues discovered from student data.

4.2 PSP Data Entry Tool

We developed a database application to support analysis of PSP data from PSP0 to PSP2, using the Progress 4GL/RDBMS [2]. In order to reduce opportunities for making mistakes, this tool was designed to require a minimum amount of user input and to provide the user with default values whenever possible. Apart from task and scheduling template values, the application automated all calculations, from determining delta times for Time Recording Log entries to performing linear regression for size estimation. In addition, the application guides the user through the appropriate forms and fields in the order most appropriate for the current process and phase.

4.3 Error Recording Method

Once the database application was ready, we entered data from the student project PSP forms manually and compared each student value with the correct value computed by the application. Although every discrepancy between the manually generated data and the application-generated data could be considered an error, we only counted an error at its insertion point. For example, in a Time Recording Log entry for the Design phase, if *Stop* is incorrectly subtracted from *Start*, *Delta Time* will be incorrect. Even if all other calculations are done correctly for the rest of the project, *Time in Phase, Design, Actual*; *Time in Phase, Total, Actual*; *Time in Phase, Design, To Date*; *Time in Phase, Total, To Date*; *Time in Phase, To Date %*; and *Time in Phase, To Date* values for an indefinite number of future projects will all be inaccurate to some degree. And this is just for the most simple process, PSP0! In more advanced processes *LOC/Hour*, time estimation, *Cost-Performance Index*, and *Defect Removal Efficiency* values could all be affected for both the current project and future projects. To eliminate this combinatorial explosion in the number of errors, we count this as a single error for an incorrect *Delta Time*. From that point on, the incorrect value would be considered correct with respect to future field values and operations.

Although we analyzed the project data quite carefully, we do not feel confident that we have uncovered all or even most of the errors in this case study. While our database application does enable us to determine the correctness or incorrectness of values generated during the analysis stage of our data quality model, it provides only limited insight into collection stage errors. For example, in the Time Recording Log, it was possible to check the *Delta Time* computation, but not the accuracy of *Date*, *Start*, *Stop*, or *Interruption Time*. Of course, the tool could not, in general, detect the absence of entries for work that was done but not recorded. Two other areas that created similar problems were the Defect Recording Log and the measured and counted *Program Size* fields for the Project Plan Summary.

4.4 PSP Error Data Analysis Tool

In order to analyze the 1539 errors uncovered by the PSP data entry tool, we developed a second database application, the PSP Error Data Analysis Tool. For each error discovered, we tracked the person who made the error, the method by which the error was found (technical review, instructor review, or comparison with the PSP tool results), the assignment in which the error occurred, the PSP process used for that assignment, the PSP phase in which the student was working when the error occurred, the general error type, the specific error type, the severity of the error, the age of the error (number of assignments since the introduction of the PSP operation in which the error occurred), the incorrect and correct values (where applicable), and an optional comment for noting issues of interest in that error.

4.5 Error Correction

Although our initial analysis of our case study data revealed many errors, the sheer presence of errors might only lead to imprecision, rather than inaccuracy. In other words, it was possible that these errors were only "noise", similar in magnitude to naturally occurring random fluctuations in behavior, but not sufficient to actually change the trends or interpretations of PSP data.

To test this hypothesis, we attempted, where possible, to fix errors so that original and corrected versions of the data could be compared. It soon became clear that errors fell into three classes. First, there were errors where the correct value could be determined. This class included such values as *LOC/Hour* that were wrong simply because of an incorrect calculation. These errors were easily fixed by correctly performing the calculation in question. Second, there were errors where the correct value could not be determined, such as a blank *Phase Injected* for a Defect Recording Log entry. Fortunately, most errors in this class occurred in fields that didn't affect other fields, such as missing header data or missing dates in the Defect Recording Log. Third, there were errors where the correct value could be guessed. In a Time Recording Log entry with *Start* 10:00, *Stop* 10:30, *Interruption Time* 0, and *Delta Time* 40; it is clear that there is a problem, but not clear which field is incorrect and should be corrected. However we can guess that there was a problem calculating *Delta Time* and assume that the other values are valid. To correct this third class of errors in an explicit and consistent fashion, we developed a set of rules. Underlying each of our rules is the assumption that primary data is more likely to be accurate than calculations performed upon it.

Rule 1: Defects in Time Recording Log entries should be handled by assuming that the start/stop/interruption times are correct and that the delta time is wrong, unless two Time Recording Log entries overlap. In that case, the preceding and following entries and the delta time for the current entry should be used to formulate plausible start/stop times. Generally this will mean starting the second entry where the first one stops.

Rule 2: If a Time Recording Log is missing an entry for an entire phase, but the Project Plan Summary form contains a value for the phase under *Time in Phase (min.), Actual*, an appropriate Time Recording Log entry should be formulated with fabricated date and time values.

Rule 3: For conflicts between a Defect Recording Log and a Project Plan Summary it should be assumed that the number of defects and the phases recorded in the Defect Recording Log are correct and that the discrepancy occurred as a result of incorrectly adding up the numbers of defects injected/fixed per phase and/or incorrectly transferring these totals to the Project Plan Summary form.

Rule 4: If, for the Defect Recording Log, the total of all fix times for defects removed in a certain phase is more than the time recorded for that phase in the Time Recording Log, a Time Recording Log entry should be inserted with start and stop times that, combined with the existing Time Recording Log entries for the phase, will produce a delta time of the total fix times plus one minute for each defect. This will represent the minimum amount of time required to find and remove the recorded defects.

Rule 5: To provide a value for a blank *Time in Phase (min.), Plan* field on the Project Plan Summary form, the value for *Time in Phase (min.), Actual* for the same phase should be used.

Rule 6: Conflicts in *Program Size (LOC)* fields on the Project Plan Summary form should be handled by assuming that *Base, Deleted, Modified, Added, and Reused* are correct and that errors are the result of incorrect calculations for *Total New and Changed* and *Total LOC*. Actually, this is not a truly satisfactory assumption because *Total LOC, Actual* should be a measurement rather than a calculation and should therefore be relied upon. However, given correct values for *Base, Deleted, Modified, Added,* and *Reused*, it is possible to calculate *Total LOC*, whereas it is impossible to even guess at the correct values for the other fields. Unfortunately, defects in the *Program Size (LOC)* fields were some of the most common defects. Combined with the importance of these fields in both size and time estimation and our inability to provide adequate corrections, estimates made with the "corrected" data were undoubtedly severely affected.

4.6 Data Comparison

After we partially corrected the project data according to the rule set, we investigated which values to compare to best reveal the effects of errors. Projects 8 and 9 had the most fields to compare since they were completed using PSP2, and provided the best opportunities for observing the cumulative effect of errors made in earlier projects. Project 9 was the best project for comparison because students had had the most practice in PSP by the time this project was completed and because it provided more time for cumulative effects to exhibit their true characteristics. Unfortunately one student did not complete this project, resulting in fewer data points for the final project.

One of the more interesting areas for comparison would have been size and time estimation. This was not possible due to the difficulties in adequately correcting the *Program Size (LOC)* fields. Instead, we selected a few fields from each of the other major sections of the Project Plan Summary, including some fields that resulted from fairly simple calculations but represented to date values from all nine projects, and other fields that were more local to the current project but were the result of more difficult operations.

5 RESULTS

Despite the discovery of data quality problems to be reported below, we still view the case study semester as an unqualified success from an educational standpoint. From a quantitative perspective, student data for the course mirrors the positive outcomes reported from other PSP case studies:

- Average defect density showed a downward trend from around 200 defects/KLOC to around 50 defects/KLOC, a 75% decrease.
- Average productivity showed a very slight positive trend, from around 15 LOC/hour to around 20 LOC/hour.
- Time and size estimation showed dramatic improvement. On the last program, both size and time estimation error dropped below 15% for half the class, with several student estimates within 3-5% of their actual values. For example, one size estimate of 507 LOC was off by only 11 LOC. One time estimate of 14.5 hours was off by only 25 minutes.
- Two students out of ten during the case study achieved what we consider to be the "Holy Grail" of PSP: 100% yield, i.e. programs that compiled and ran correctly the first time without any syntax or run-time errors.

The qualitative outcomes were equally positive. Most students expressed a very high degree of satisfaction with the course, for example:

- "I think PSP is one of the few things I learned in this school that is useful. It will be definitely useful on my job."
- "I think PSP should be a required class for graduate students and maybe a elective class for undergraduate students."
- "In September, I didn't know anything about software engineering. Now I know a great deal thanks to PSP. I now know the importance of why a process is used to finish a task. Software development is not easy and using a process helps in development. I have learned that my design skills aren't that great but my debugging skills is (sic) pretty good."
- "I thought I was a good programmer, but after using PSP I realized that I was nothing back then. Now, I can proudly say that I have gotten much much better than ever before."
- "I must admit, when I started this course, I understood what we were supposed to do in good software engineering, but I never really did it. Now I understand the reasons behind these practices and the benefits of actually following a process instead of just jumping right into coding. Learned: 1.) Defect prevention and removal is an extremely important skill. 2.) My design skills need serious work. 3.) To be able to make an estimate and be reasonably confident of its accuracy is very useful. 4.) Teachers who push doing planning and design might actually know what they're talking about."
- "At the beginning, I just coded to finish the project or solve the problem. Now I take an in-depth look at the problem and think about it for a while before trying to develop a solution. By executing and learning this process I know way more about software engineering than when I started this course."

Despite modifications to the PSP curriculum to increase data quality, our analyses yielded 1539 data errors. In this section, we report on: the types of errors found; their severity; their age; the manner in which they were detected; analysis stage errors; collection stage errors; and error impact.

5.1 Error Types

We found that the errors naturally fell into one of seven general types. We present each type in descending order of frequency, and include the number of errors found of that type and the percentage of all errors represented by this type.

Calculation Error. (705 errors, 46%). This error type applied to data fields whose values were derived using any sort of calculation from addition to linear regression. If the calculation was not done correctly, an error was counted. This type was not used for values that were incorrect because other fields used in the calculation contained bad numbers.

Blank Field. (275 errors, 18%). This error type was used when a data field required to contain a value, such as the *Start* field in a Time Recording Log entry, was left blank. This type was not used in fields where a value was optional, such as comment fields.

Transfer of Data Between Projects Incorrect. (212 errors, 14%) This error type was used for incorrect values in fields that involved data from a prior project. Typically these fields were "to date" fields that involved adding a to date value from a prior project with a similar value in the current project. Unfortunately, it was often impossible to determine if the error arose from bringing forward a bad number, or incorrectly adding two good numbers, or bringing forward the correct number and correctly adding it to the wrong number from the current form. However, in two important areas, time and size estimation, the forms were modified so that students were required to fill in the prior values to be used in the estimation calculations. In these cases we could determine when incorrect values originated in the transfer.

Entry Error. (142 errors, 9%). This error type applied when a student clearly did not understand the purpose of a field or used an incorrect method in selecting data. Examples include the use of a phase name in the *Fix Defect* field of the Defect Recording Log, or having the *Defects Injected, To Date* values in the Project Plan Summary originate from a different project than the *Program Size (LOC), To Date* values.

Transfer of Data Within Project Incorrect. (99 errors, 6%). This error type is similar to the error type involving incorrect transfer of data between projects, except that it applied to values transferred from one form to another within the current project. For example, filling in 172 for *Estimated New and Changed LOC* on the Size Estimating Template, but using 290 for *Total New and Changed, Plan* on the Project Plan Summary.

Impossible Values. (90 errors, 6%). This error type indicates that two values were mutually exclusive. Examples of this error type include overlapping time log entries, defect fix times for a phase adding up to more time than the time log entries for the phase, or phases occurring in the Defect Recording Log in a different order than those in the Time Recording Log.

Process Sequence not Followed (16 errors, 1%). This error type was used when the Time Recording Log showed a student moving back and forth between phases such as Compile and Test instead of sequentially moving through the phases appropriate for the process.

5.2 Error Severity

Some PSP data errors have relatively little "ripple effect" upon other data values, while others can have an enormous impact. To gain insight into the distribution of the ripple effect, we classified the errors into one of five "severity" levels. We present the levels in increasing order of ripple effect. As before, we include the total number of errors found for a given severity level and its percentage of the total.

Error has no impact on PSP data. (104 errors, 7%). This level included errors such as missing header data, incorrect dates in the time recording log, and filling in fields for a more advanced process.

Results in a single bad value, single form. (674 errors, 44%). This level was used if a significant field which affected no other fields, such as *LOC/Hour, Actual*, was blank or incorrect.

Results in multiple bad values, single form. (197 errors, 13%). This level indicates when an incorrect or blank value was used in the calculation of values for one or more other fields on the same form, but when none of these other values were used beyond the current form. For example, in PSP1 on the Size Estimating Template, incorrectly calculating a prediction interval. This results in a bad prediction interval and a bad prediction range, but these values are not used anywhere else in the process.

Results in multiple bad values, multiple forms, single project. (41 errors, 3%). This level indicates when an incorrect or blank value was used to determine the values for one or more other fields on one or more different forms in the same project, but when none of these other values were used beyond the current project. For example, in PSP1, on the Size Estimating Template, calculating an incorrect value for *Estimated Total New Reused (T)*. This results in an incorrect value for *Total New Reused, Plan* on the Project Plan Summary form, but this value is not referenced by future projects.

Results in multiple bad values, multiple forms, multiple projects. (523 errors, 34%). This level was used if an incorrect or blank value affected future projects. For example, when *Defects Injected, Planning, Actual* on the Project Plan Summary does not match the number of defects entered for the planning phase in the Defect Recording Log.

5.3 Age of Errors

In any learning situation, a certain number of errors are to be expected. We hypothesized that perhaps the errors we discovered were simply a natural by-product of the learning process, and would "go away" as students gained experience with the various techniques in the PSP.

To evaluate this hypothesis, we calculated the "age" of errors—in other words, the number of projects since the introduction of the data field in which the error could be observed. If the errors were simply a by-product of the learning process, then we would expect a low average "age" for errors (people might make an error in a field initially, but then stop making the error after gaining more experience with the data field in question.

For example, the calculation of *Delta Time* for the Time Recording Log was introduced in the first project. If a student made an error in this field during the first project the error would have an age of zero. If a similar error was made during the second project the error would have an age of one. By the ninth project this type of error would have an age of eight.

We first analyzed the errors to determine the average error age in each project. Figure 2 shows the average age for all errors in each project.

We then filtered out the 309 errors with an age of zero. This eliminated errors that could result from students being introduced to new fields and/or PSP operations for the first time. Figure 3 shows the resulting data.

When combining the 1539 errors from all projects, the average error age was 2.78 projects. After removing the 309 errors with an age of zero, the average error age rose to 3.48 projects.

5.4 Error Detection Methods

In this study, there were three ways an error could be detected: by another student during technical review (40 errors), by the instructor during the grading/evaluation process (32 errors), or through the use of the PSP data entry tool (1467 errors). Thus, students were

Project #	PSP Process	# of Errors	Average Age
1	PSP0	51	0.00
2	PSP0.1	59	0.73
3	PSP0.1	63	1.76
4	PSP1	150	1.27
5	PSP1	165	2.27
6	PSP1	186	3.30
7	PSP1.1	160	3.26
8	PSP2	351	3.04
9	PSP2	354	3.84

Figure 2: Average Error Age by Project - All Errors

Project #	PSP Process	# of Errors	Average Age
1	PSP0	0	NA
2	PSP0.1	43	1.00
3	PSP0.1	63	1.76
4	PSP1	70	2.71
5	PSP1	165	2.27
6	PSP1	186	3.30
7	PSP1.1	135	3.86
8	PSP2	214	4.99
9	PSP2	354	3.84

Figure 3: Average Error Age Where Age is not Zero

made aware of about 5% of the mistakes in their completed projects during the course of the class.

5.5 Analysis Stage Errors

Our two stage model of PSP data quality indicates that errors can be introduced during either collection or analysis. Most of the errors that we detected occurred during PSP analysis activities, with 700 errors occurring in the Plan phase and 561 errors in the Postmortem phase. Some of the errors occurring in other phases, such as errors in *Delta Time* calculations, were also analysis errors.

5.5.1 The Most Severe Errors

34% of errors found were of the most serious type - persistent errors. These were the errors resulting in multiple bad values on multiple forms for multiple projects. A defect of this type not only causes incorrect values in the current project, but may still be causing flawed results ten projects later, even if all subsequent calculations are done correctly. Figure 4 shows the four most common errors of this type.

Description	#
Time Estimation: historical data not transferred correctly	61
Size Estimation: historical data not transferred correctly	56
Time Log: delta time incorrect	48
Project Plan Summary: Total LOC, actual, not equal to B-D+A+R	45

Figure 4: Most Frequently Occurring Persistent Errors

Description	Errors	Time Log Entries	% in Error
Project 1	7	84	8.33
Project 2	2	88	2.27
Project 3	8	92	8.70
Project 4	8	108	7.41
Project 5	2	102	1.96
Project 6	9	121	7.44
Project 7	2	77	2.60
Project 8	5	122	4.10
Project 9	5	105	4.76

Figure 5: Delta Time Errors by Project

There were two main ways that the error in transferring time estimation data appeared to occur: incorrectly transferring the value from the correct field, or accidentally transferring the correct value from an incorrect field. For example, instead of transferring *Total New and Changed (N)* (Plan or Actual), students often transferred *Total LOC (T)*. This could easily occur because the Project Plan Summary form has over 90 fields even at the level of PSP1, and these two values are vertically adjacent on the form. It is particularly easy to make this mistake with the Actual values because the fields are separated by one column from the labels. Additionally, it appeared that students made spreadsheets to avoid thumbing through the entire stack of completed projects every time a time or size estimation was needed for a new project. We infer this because the same incorrect value for a particular project would be transferred over and over again for time and/or size estimation in new projects.

Similar factors surrounding the error in transferring data for size estimation. These transfer errors were not insignificant. Over the 56 errors resulting from incorrect transfer of data used for size estimation, the sum of the errors was 7753 LOC (lines of code), with an average error of 138.4 LOC. The sum of the LOC as they should have been transferred was 10,255, with an average of 183 LOC per field. Thus, the average incorrectly transferred number was in error by an amount equaling 75.6% of the number that should have been transferred.

The error in calculating *Delta Time* in the Time Recording Log was notable in several respects. First, the errors were not insignificant. The average mistake was 37.8 minutes, which was an average of 39.9 percent of the correct value. Secondly, of 48 occurrences, 16 were in error by one hour and 4 were in error by two hours, indicating small errors in simple arithmetic. Thirdly, the distribution of this error across projects is as shown in Table 5.

Despite nine projects worth of experience, this error never "went away". However it did appear to occur less frequently after Project 6. Interestingly, the assignment for this project was a Time Recording Log applet, which at least some students seem to have used for subsequent projects.

5.6 Collection Stage Errors

As noted previously, analysis stage errors are relatively easy to determine and correct. However, the accuracy of recorded process measures from the collection stage was much more difficult to examine because the time of collection had already passed and, unlike the analysis operations, was impossible to reproduce. However, we found both direct and indirect evidence for collection errors during the case study.

5.6.1 Direct Collection Error Evidence

Direct evidence of collection problems appeared in the 90 errors of type of "Impossible Values". We classified these errors into three major subtypes.

Internal Time Log Conflicts. There were five time logs with overlapping entries, indicating some sort of problem with accurately collecting time-related data.

Internal Defect Log Conflicts. 51 errors showed problems with correctly collecting defect data. 48 of these errors were Defect Recording Log entries showing defects injected during the Compile and Test phases, but not as a result of correcting other defects found during Compile or Test.

Discrepancies Between Time and Defect Logs. In 22 cases, Defect Recording Log entries were entered with dates that did not match any Time Recording Log entries for the given date. For example, a defect would be recorded as injected during the Code phase on a Wednesday, but the time log would show that all coding had been completed by Monday and that the project was in the Test phase on Wednesday. For 10 projects, the total *Fix Time* for defects removed during a particular phase added up to more time than was recorded for that phase in the Time Recording Log. Finally, in two cases, the Defect Recording Log showed a different phase order than the Time Recording Log.

5.6.2 Indirect Collection Error Evidence

Besides the recorded errors, there were other indicators that collection problems had occurred. Some Time Recording Logs showed a suspicious number of even-hour (e.g. 6:00 to 7:00, 10:00 to 12:00) entries. Others showed long stretches of consecutive entries with no breaks or interruptions. Often, the total *Fix Time* for the defects in a phase was far less than the time spent in the phase. For example, the Time Recording Log might show three hours spent in the Test phase, but the Defect Recording Log would show two defects that took eight minutes to fix. Obviously, it is not impossible that this would occur, but it is much more likely that not all defects found in test were recorded.

In a similar vein, some projects had suspiciously few defects overall, such as seven defects for a project with 284 new lines of code and almost 11 hours of development time, (including 40 minutes in compile for two defects requiring 6 minutes of fix time). Our analysis of the PSP data for that same project yielded 27 errors.

Finally, the instructor has anecdotally observed the following trend in every PSP course he has taught so far: the students turning in the highest quality projects also tend to record far higher numbers of defects than the students who turn in average or lower quality projects. If this trend is real, then we can provide two possible explanations. It may be the case that the students turning in lower quality projects tend to make far fewer errors than those turning in the higher quality projects, although this seems *extremely* unlikely. What appears more likely is that the students turning in the highest quality projects also exhibit the lowest level of collection error, which indicates that substantial but non-enumerable collection error exists in the PSP data we examined.

5.7 Comparison of Original and Corrected Data

When we compared the original and corrected data, we found significant differences ($p_{\dot{c}}.05$) for the Cost-Performance Ratio (planned time-to-date/actual time-to-date) and Yield (percentage of defects injected before first compile that were also removed before first compile). We used the Wilcoxon Signed Rank Test [**?**], a non-parametric test of significance which does not make any assumptions regarding the underlying distribution of the data. Figure 6 and Figure 7 illustrate the differences between these two measures graphically. Comparison of original and corrected data for other measures did not yield such significant differences.

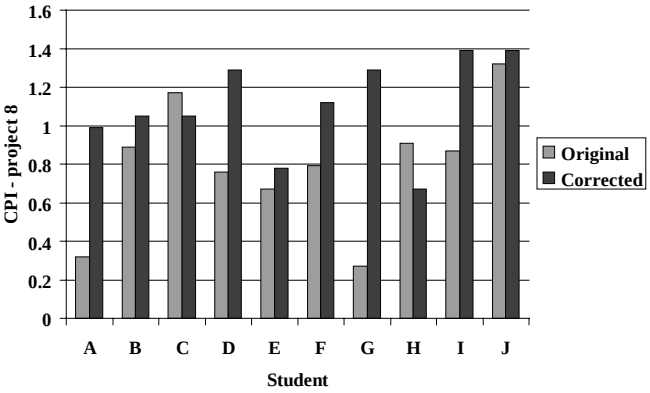

Figure 6: Effect of Correction on CPI

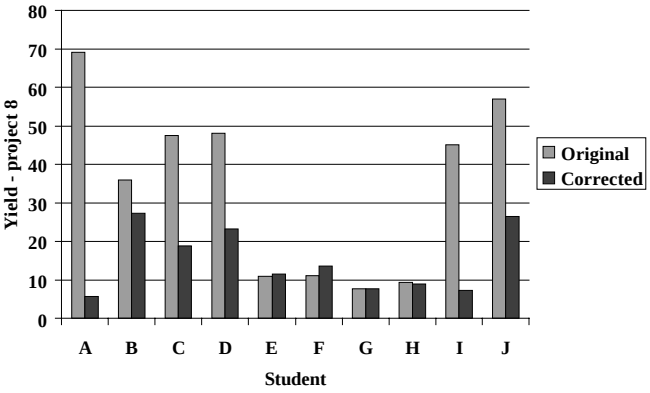

Figure 7: Effect of Correction on Yield

A CPI value of 1 indicates that planned effort equals actual effort. CPI values greater than 1 indicate overestimation of resource requirements, while CPI values less than 1 indicate underestimation of resource requirements. In half of the subjects, correction of the CPI value reversed its interpretation (from underplanning to overplanning, or vice-versa). In the remaining cases, several corrected CPI values differed dramatically from original values. For example Subject A's original CPI was 0.32, indicating dramatic underplanning, while the corrected CPI was 0.99, indicating an average planned resource requirements virtually equal the average actual resource requirements.

Correction of yield values tended to move their values downward, sometimes dramatically. In half of the subjects, the corrected yield was less than half of the original yield values, indicating that subjects were removing a far fewer proportion of defects from their programs prior to compiling than indicated by the Yield measurement.

6 CONCLUSIONS

This paper reports on the results of analysis of the data from a single PSP class with only 10 students. As with any case study, care must be taken in interpreting these results. We do not know whether this data is representative of PSP courses in general, and if the way we teach the PSP is representative of the way the PSP is taught by others.

While we do not claim that these results are representative of all PSP courses, neither do we believe that they result from some peculiarity and/or failing of our environment. First, this case study was performed after the instructor had taught the PSP for one semester in a graduate level course, and instituted it within his research group, and adopted it himself for his own software development activities. By the time of this study, we were quite experienced as both teachers and users of the PSP. Second, as already noted, we were concerned with data quality problems from the beginning, and instituted curriculum modifications specifically intended to raise data quality. Third, our results cannot be due to our lack of enthusiasm for the PSP: both of us consider it to be one of the most powerful software engineering practices we have acquired in our careers. Fourth, our results cannot be due to lack of enthusiasm for the PSP by our students: in the post-course evaluation, most of the students indicated that they found the class to be very useful and interesting.

We believe there are two fundamental conclusions to be drawn from this case study. First, we believe this study indicates the need to initiate efforts to explicitly assess collection and analysis data errors by others using the PSP in both industrial and academic settings. With better understanding of these two types of errors and their impact upon the PSP, the community can better guide the evolution of the PSP toward higher data quality.

Second, we believe that until questions raised by this study with respect to PSP data quality can be resolved, *PSP data should not be used to evaluate the PSP method itself*. In other words, we believe that it is not yet appropriate to infer that changes in PSP measures during (or after) a training course accurately reflect changes in the underlying developer behavior. A statement such as "The improvement in average defect levels for engineers who complete the course is 58.0%", if based upon PSP data alone, might only reflect a decreasing trend in defect recording, not a decreased trend in the defects present in the work product.

We are happy to report that not all PSP evaluations are based upon PSP data alone. For example, in one of the case studies [3], evidence for the utility of the PSP method was based upon reductions in acceptance test defect density for products subsequent to the introduction of PSP practices. Although alternative explanations for this trend can be hypothesized (such as the PSP-based projects were more simple than those before and thus acceptance test defect density would have decreased anyway), at least the evaluation measure is independent of the PSP data and not subject to PSP data quality problems.

We conclude this report with the following conjecture: substantial, integrated automated support for both collection and analysis stages in the PSP is *essential*, not merely *helpful*, if high data quality is desired. Automation of the analysis stage is not a new idea to the PSP community, and many spreadsheet and database tools have already been made available. However, although the case study class had spreadsheets, Java programs, and other tools available to them, we surmise that they were not sufficiently integrated with each other and with the forms to prevent data quality problems from occurring. When contemplating PSP automation, however, it is wise to remember the age-old computing axiom: "garbage in, garbage out". No matter how automated the analysis stage, overall PSP data quality will still depend completely upon the data quality from the collection stage. Integrated, automated support for both collection and analysis stages in the PSP is a challenging goal for future research on personal software process improvement.

References

[1] Anna Ch. Ceberio-Verghese. Personal software process: A user's perspective. In Nancy R. Mead, editor, *Ninth Conference on Software Engineering Education*, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, CA 90720-1264, April 1996. IEEE Computer Society Press.

[2] Progress Software (Data Language Corporation). Information is available at: www.progress.com/core/develop.htm.

[3] Pat Ferguson, Watts S. Humphrey, Soheil Khajenoori, Susan Macke, and Annette Matvya. Introducing the personal software process: Three industry cases. *IEEE Computer*, 30(5):24–31, May 1997.

[4] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, 1993.

[5] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, New York, 1995.

[6] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[7] Watts S. Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley, New York, 1997.

[8] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[9] James E. Tomayko. Carnegie mellon's software development studio: a five year retrospective. In Nancy R. Mead, editor, *Ninth Conference on Software Engineering Education*, 10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, CA 90720-1264, April 1996. IEEE Computer Society Press.