DATA QUALITY PROBLEMS IN THE PERSONAL SOFTWARE PROCESS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAI‘I IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

December 1998

By

Anne M. Disney

Thesis Committee:

Philip M. Johnson, Chairperson
James Corbett
Martha Crosby

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

To

Gertrude Raven

Acknowledgments

To Jeanette Teehee, Kim Carr, Cyndee Grady, Dad, and Mom: Your kind words of encouragement have made the last two years much easier.

To Jim Wuerstlin, Bryan Sakka, Dad, Gayle Johnson, and Clare Joslin: You actually wanted to know what my research was about!

To Charles: You dropped me off at the airport and picked me up again every week over three semesters, and didn't make me hitchhike even once!

To my friends at CSDL: Your support has been invaluable. You all have given me useful feedback as you critiqued my presentations and reviewed my writing. Robert, your advice has been helpful on many subjects. Cam, I've asked you many, many questions on a wide variety of topics; you've almost always known the answer and have never made me feel like an interruption or annoyance. Jen, need I say more than "ICS 613" or "Probability, Statistics, and Queuing Theory"? Yes, actually, I do: Thank you for allowing me to use your PSP data in this research and for being a steady, hardworking partner through all those late nights and early mornings.

To everyone at Infoworld Management Systems; particularly Jim Sullivan, Linda Augustine, and Laura Croce: You have never complained about my unpredictable schedule and curtailed working hours. You never asked, "Are you ever going to get done?" or urged me to finish faster. I appreciate your tremendous flexibility while I have pursued this goal.

To Cyndee and Denny Grady: Without your friendship and your generous offer of a place to stay, I could not have continued after moving from O'ahu. There are really no words for friends like you. And to Nicole: you're too young to understand this yet, but your joyful cries of “Auntie *Anne* is home!” cheered me up many times after a tiring day at school.

Finally, to Philip Johnson: Your enthusiasm gave me a vision of what I could accomplish. Your expertise and world-class teaching showed me how to do it. Your confidence in me has made everything possible. Thank you!

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 me to wonder about the quality of two areas of PSP practice: collection and analysis. To investigate this I built a tool to automate the PSP and then examined 89 projects completed by nine subjects using the PSP in an educational setting. I discovered 1539 primary errors and analyzed them by type, subtype, severity, and age. To examine the collection problem I 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 I developed a rule set, corrected the errors as far as possible, and compared the original and corrected data. This resulted in substantial differences for numbers such as yield and the cost-performance ratio. The results raise questions about the accuracy of published data on the PSP and directions for future research.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

CMM	Capability Maturity Model
KLOC	1000 Lines Of Code
LOC	Lines Of Code
PIP	Process Improvement Proposal
PSP	Personal Software Process
TSP	Team Software Process

Chapter 1

Overview

1.1 Why is Software Quality Important?

Between 1985 and 1987 two people died and four were seriously injured when they received massive radiation overdoses delivered by the Therac-25 radiation therapy machine. Subsequent investigations uncovered many complex factors leading to these accidents, including poor user interface design, problems with race conditions, and other software bugs [17].

The Denver International Airport officially opened in March 1995 - 16 months late and more than 100 million dollars over budget. One major reason was the infamous automated baggage handling system. Political, mechanical and electrical problems all contributed to its failure, but the presence of major bugs in the controlling software was a primary factor[7].

In June 1996 the maiden flight of the Ariane 5 rocket launcher failed about 40 seconds after lift-off. The unmanned rocket veered off course and automatically self-destructed, completely destroying both the rocket and the four scientific satellites on board. The total cost was about 2.5 billion dollars. The immediate cause was traced to a software error in converting a 64-bit floating point number to 16 bit signed integer [15]. However, the report by the Inquiry Board goes deeper and blames this error on “specification and design errors in the software” and inadequate analysis and testing of the failed subsystems [18].

1.2 Tackling the Problems

As these recent incidents illustrate, producing high-quality software on schedule is a major issue in the computer industry today. However, this is nothing new - ever since the beginning of computer programming, bugs and behind-schedule projects have been persistent problems.

Over the decades, the software industry has taken various approaches to addressing these challenges. Into the 1960's, there was a general feeling that programmers just needed to "try harder". In the mid-70's, the importance of testing was emphasized. Eventually it became clear that although testing is very important, it is only a partial solution since it is impossible to test exhaustively. In fact, this gave rise to the software engineering adage that, "You can't test in quality." By the mid-80's the software engineering community began to realize that without a high-quality process it is difficult to produce high-quality software, especially for large or complex projects. Standards such as the ISO 9000 for organizations and the Capability Maturity Model (CMM) for organization-wide software processes were developed. In the end, however, high-quality software is produced by individuals. These organization-level processes are, not surprisingly, more helpful to organizations than individuals. In the mid-90's the focus has begun to shift back to the individual developer with the introduction of the Personal Software Process (PSP) [9].

1.3 The Personal Software Process

The PSP is a self-improvement process for programmers. In effect, each programmer performs a longitudinal experiment in which there is only one subject. Each new program written can benefit from data collected about past projects, and can in turn provide new insights to improve planning, productivity, and quality for future work. But the programmer must adhere to a rigorous and complicated process to make this happen.

The PSP has two main goals. The first is to produce high-quality programs as efficiently as possible. By paying close attention to every defect made as well as the overall patterns of errors, PSP users become more aware of quality issues. The PSP provides them with structured mechanisms such as code review that can help them to prevent similar defects in future projects. Additionally, by tracking when and where each defect is injected and removed and the time required for removal, developers are continually reminded that finding and removing defects early in the process is much less time-consuming and less costly than waiting to remove defects during testing.

The second goal of the PSP is to improve accuracy in time, size, and quality estimation. If the size of a program can be accurately predicted, then the developer can often make a good estimate of the total time required for completion. This, of course, improves the quality of schedules and budgets. Programmers do size and time estimation at the beginning of each project. As soon as they complete some preliminary planning, they select a similar set of past projects. Then they use various statistical techniques to compute likely size and time values for the current project.

When a programmer uses the PSP, he or she starts by doing planning work using a set of predefined worksheet-type forms. Then, while moving through the design, code, compile, and test phases, he or she uses other forms to keep detailed records about the time spent in each phase and defects injected and removed. When the project is completed there is is a final "postmortem" phase in which the programmer analyzes all the data collected about the project and computes values like *Lines of Code (LOC) per Hour* and *Defects per Thousand Lines of Code (KLOC)*. He or she calculates such values for the current project; and then calculates them again as "to date" values, including not only the current project, but the entire set of similar projects used in planning. Therefore, at the most basic level, the PSP involves two main activities: collecting primary data such as size, defect, and time measures; and analyzing this data to produce derived or to-date measures.

1.4 Motivation for this Research

Humphrey and others provide data from the use of PSP in a classroom setting that shows positive trends in various measures of the software process, including defect density, yield (percentage of defects actually removed), and time estimation. For example, a 1997 study done at Carnegie-Mellon University showed a reduction of size estimation error by a factor of 2.5, a reduction of time estimation error by a factor of 1.75 or more, and a dramatic reduction in the number of defects injected [8]. Unfortunately, most of the data used to illustrate these positive trends about the PSP is actually data produced by the PSP itself.

In January 1997 I had been using the PSP for a full year for both academic and professional work. Dr. Philip Johnson (my thesis advisor) was also using the PSP and had twice taught a full-semester class on the PSP. Generally, the standard good results of using the PSP had been replicated in his classes. However, we were aware by this time of the complexity of the PSP, the difficulty of some of the calculations, and the personal commitment required to do a good job of collecting primary data. We began to wonder just how many errors people made while using the PSP and what sort of effect these errors would have on the measures produced.

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 1.1. The model begins with “Actual Work”—the 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 into ways to improve future software development activities – the “Insights about Work”.

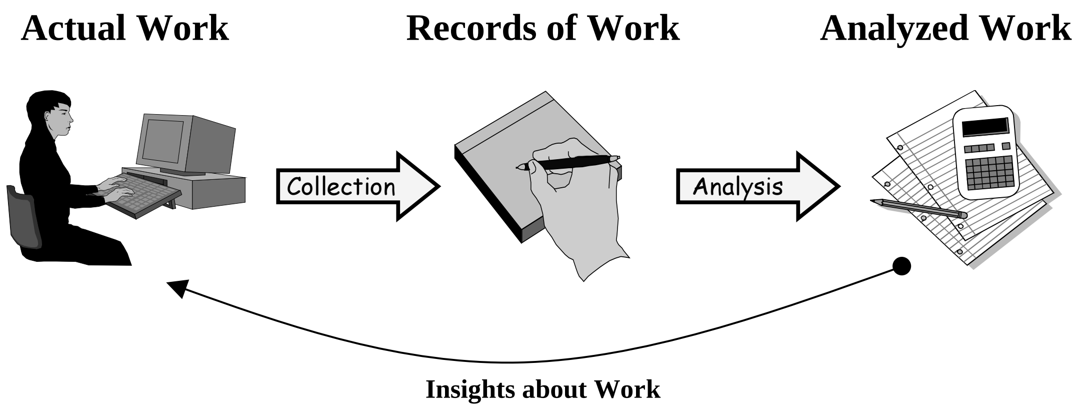


Figure 1.1: *A simple model for PSP data quality. Through a process of* collection*, the developer generates an initial empirical representation ("Records of Work") of his or her personal process ("Actual Work"). Through additional* analyses*, the developer augments this initial empirical representation with derived data ("Analyzed Work") intended to enable process improvement using "Insights about Work".*

We based this model upon the PSP as presented in *A Discipline for Software Engineering* [9] and as practiced by the students in Dr. Johnson's classes - what we termed "manual PSP". This means that although students had various helping tools ranging from calculators to spreadsheets to a size estimation applet, they had to fill out all forms by hand and had no computerized guidance in following the outlined sequence of process steps. In contrast, what we termed "automated PSP" is done using some kind of software package that performs all possible calculations for the user, inserts and maintains the correct calculated values in the appropriate places in the forms, guides the user through the process steps, and aids in the collection of primary data. Note that although an "automated" PSP can essentially automate all of the analysis state 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 Dr. Johnson taught this class, there was no integrated software support for the PSP. 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.)

1.5 Hypothesis of this Research

From this model, we isolated two areas that could have a negative impact on PSP data quality: collection and analysis. Unless the collected data reflects the actual behavior of a programmer, the derived measures will be based upon an inaccurate model of the work done. Furthermore, analyses performed on the collected data must be done correctly in order to provide meaningful insights into the programming process.

Therefore, our model of the PSP led to the following two hypotheses:

1. The PSP suffers from a collection data quality problem.

2. Manual PSP suffers from an analysis data quality problem.

1.6 Case Study

In order to test these hypotheses, we decided to do a case study to evaluate PSP data quality, using data collected from the next PSP class taught by Dr. Johnson. We wanted to determine what kinds of errors were made and how often they occurred. Of course, no human being can do a perfect job at all times of either collection or analysis – a certain number of errors is to be expected. Therefore, we felt it was important to evaluate not only the numbers and kinds of errors, but their effect on important derived measures. Given the nature of the data, we knew it would be impossible to correct all errors with complete accuracy, but we wanted to see if even partial correction was enough to observe any substantial differences between original and corrected data.

Even before teaching the PSP class that produced the projects evaluated by this case study, Dr. Johnson had concerns about the quality of PSP data. Therefore, he made four main modifications to the standard PSP curriculum: (1) increased process repetition, (2) added four worksheets to help students through the most difficult analyses, (3) implemented in-class technical reviews, and (4) provided tool support.

Once the class was completed and all the projects were available for review, I wrote two database applications: one to automate a large part of the PSP and another to track errors made while using the PSP. Then I took the 90 software projects (9 projects each by 10 students) completed during the PSP class and entered all the primary values into the PSP application. I then compared every derived value on the hand-completed forms with the values generated by the computer. Whenever there was a discrepancy, I recorded the error in the error-tracking tool. (However, I only recorded errors at their source. For example, if a derived measure was incorrectly calculated, and then was used itself to produce other derived measures, I only counted one error even though multiple derived measures were incorrect.) Finally, I formulated a set of correction rules, attempted to repair all the detected errors, and then compared the original and corrected values for some important derived measures.

1.7 Results

When this research project started, Dr. Johnson and I expected to find between 50 and 100 errors in the class data. This estimate was way off: we found 1539 errors. When I analyzed the errors by type, the most common error types were errors in calculations, blank fields, and inter-project transfer errors. There were also 90 errors that indicated deeper problems in the collection of primary data measures.

Another way of classifying errors was by severity. In other words, was an incorrectly calculated value isolated, or was it used in other calculations, thereby corrupting other fields? If it did corrupt other fields, were the errors confined to the current form or project, or were the fields used by calculations in future projects? 44% of the errors were confined to a single bad value on a single form. However, 34% of the errors resulted in multiple bad fields on multiple forms for multiple projects.

One could argue that many errors were made simply as 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, I assigned each error an “age” corresponding to the number of projects since the introduction of the field in which the error occurred. When looking at all errors, the average age was 2.8 projects. When looking only at errors with an age greater than 0, the average age was 3.5 projects.

One might also hypothesize that these errors simply constitute “noise” and do not substantially impact upon the overall analysis results. To test this hypothesis, I recalculated some of the major metrics using the (partially) corrected data. I found clear differences for such measures as *Yield* and *Cost-Performance-Index*. When looking at the eighth project, these values were substantially affected for at least half the students.

1.8 Implications

This case study shows that PSP users can make substantial numbers of errors in both the collection and analysis stages, and that these errors can have a clear impact upon measures of quality and productivity. However, these results do not imply that the PSP method is wholly unuseful in improving time estimation and software quality. Instead, these results could be useful in motivating two essential types of improvement to the PSP process: attention to measurement dysfunction issues and integrated automated tool support. 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.

1.9 Organization of this Document

Chapter Two will provide a summary of the Personal Software Process, including its goals and the methods used to achieve them.

Chapter Three takes a look at related works, focusing on published results about the PSP, automation of the PSP, human error, and measurement dysfunction.

Chapter Four will describe the case study - how I evaluated the PSP data from 90 software projects, the rules I exercised to correct errors in the PSP data, and the method I used to record results.

Chapter Five will describe the two software tools I wrote to automate the PSP and to record and analyze the errors I found while evaluating the student projects.

Chapter Six will present the results of the case study, including summaries of error types, error severity levels, age of errors, error detection methods, and effects of data correction. It will also give a closer look at the most severe errors.

Chapter Seven will cover the conclusions of this research and explore ideas for future research.

Chapter 2

The Personal Software Process

When elite athletes work on their skills, they often use an execute-evaluate-change cycle. For example, a diver first performs a reverse somersault with 2-1/2 twists while his coach records it on videotape. Then together they play the tape and evaluate the dive for certain key points such as take-off, extension, and amount of twist. Subjectively, the dive felt pretty good, but the diver sees that his body position just before entering the water was poor. His coach points out that to correct this he needs to straighten his knees sooner and point his toes harder. The diver then tries again (and again and again!)

Elite programmers also want to improve their skills and could certainly benefit from a similar learning cycle, but software development is very different from diving! We do execute work, but evaluating it is much more complex than watching a videotape. To begin with, software development is much too time-consuming, complex, and intellectually driven to make any direct recording mechanisms useful. These same factors also make it difficult to "replay" the software development experience mentally - too much has been experienced. Finally, very few developers have what is considered essential in athletic endeavors - a coach. A coach has years of experience in the field, provides objective insights, constantly watches and evaluates learning and practice sessions, distinguishes problems from symptoms, and assesses weaknesses. Most importantly, a coach has a specific vision of what the athlete can eventually accomplish and understands the long process of incremental steps that leads to the final goal.

This is not to say that learning to be a better software engineer is completely different from learning to be a better diver. First, in both areas, the subjective and objective experience of the same event can be very different. A dive can feel pretty good to a diver because of a clean entrance into the water, but look sloppy to an experienced observer. Similarly, a developer may feel that development for a particular program was relatively smooth and that the final product is good, but forget about an initial time estimate that was 30% too low and be completely unaware of several hours wasted fixing bugs in test that could have been prevented by more thorough design. Second, in both areas there is a gray scale for evaluation. Diving is not like football where players clearly (most of the time) either do or do not make a touchdown. Just because a diver makes it to the water doesn't mean he did it well! Similarly, software development has no single, objective metric that enables us to rank one development effort over another. Does one use on-time completion, user satisfaction with the final product, efficiency of development, reusability of the code, elegance of the design, quality of the documentation, or quality of comments in the code? In the end it is a subjective decision based on a complex evaluation of multiple criteria. Finally, in both areas, the attitude of the person trying to improve is a key factor. Parents can send a child to the most expensive diving club with the best coach and force the child through five hours of practice a day, but without a desire to succeed and willingness to put forth the effort to get the most out of each repetition and conditioning exercise, the young diver will never stand on the Olympic podium. Similarly, without a desire to improve and a willingness to work at it, even a good brain, a time-tested process, and the most expensive development environment cannot produce a truly accomplished software engineer.

So what does this have to do with the PSP? For a motivated programmer, the PSP can produce, in a virtual sort of way, a videotape of the development process. The structured forms and processes of the PSP provide the programmer with quite a bit of data about each step taken. Later, this distilled model of the development process can be looked at from many angles. It will not change with the passage of time, as a memory of the process would. Also, although nothing can replace a human mentor, the PSP provides a constantly available "coach" for the developer. In some areas, the robust statistical techniques involved help the "coach" to make authoritative statements such as, "Over the past

20 projects you have shown a tendency to underestimate the required time by about 30%. Based on your estimate for this new project at 200 minutes, you should probably count on actually needing about 280 minutes to finish." In other areas, the "coach" can only point out deficiencies such as, "You making 181 errors per thousand lines of code. 64% of these errors are injected in design and 80% are being removed in test." Although the PSP does provide coach-like guidance via a structured set of increasingly complex processes designed to support better and better software development, the developer is responsible for finding the cause of weaknesses like this and deciding on the specific steps needed for correction so that the next project can be better done.

In essence, the PSP provides a software engineer with the tools necessary to improve using an execute-evaluate-change cycle: Work is done as the developer uses the PSP to record it, the PSP provides a specific structure that can be used to objectively guide subjective analysis, and the PSP insights combined with developer experience illuminate areas that need improvement and the specific steps required to improve. Attitude is still a key factor - a sloppy or unwilling PSP user can distort the "videotape", garble the comments of the "coach", or go through the motions without really implementing changes which would lead to improvement.

2.1 PSP Levels

The PSP is really a set of increasingly complex processes. These are summarized in Figure 2.1. When learning the PSP, a software engineer starts at PSP0 and gradually learns the more advanced processes one at a time. Only the first five processes were used in the PSP class which produced the projects reviewed for this research. Before reviewing them in more detail, it is important to understand the PSP is based on a unidirectional phase model, where software development occurs in series of phases (such as design, code, compile), and once a phase is completed the development cannot return to that phase again (except

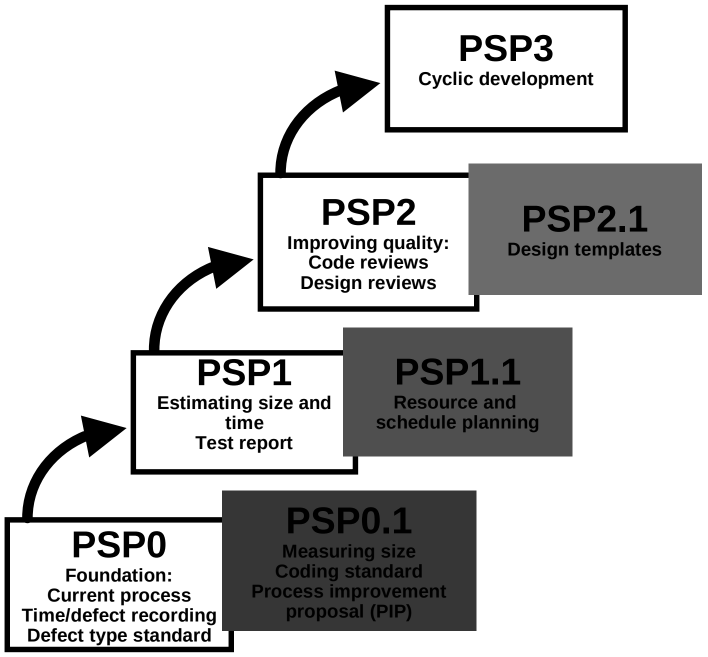


Figure 2.1: *PSP Overview*

for PSP level 3 which is designed for large projects).

2.1.1 PSP0: Foundation

This baseline version of the PSP provides an introduction to the PSP environment of processes, forms, instructions, and process scripts. It introduces six development phases: planning, design, code, compile, test, and postmortem. It uses three main forms: Time Recording Log, Defect Recording Log, and PSP0 Project Plan Summary. In the planning phase, the user produces or obtains a requirements statement for the project, verifies that it is clear, and makes a “best-guess” estimate of the amount of time the project will require. Throughout the design, code, compile, and test phases, the user records time and defect data using the Time and Defect Recording Logs. Finally, in the postmortem phase, the user extracts information from these logs and from prior projects (after the first project) to pro-

Table C14a PSP0 Project Plan Summary

Student	Jill Fonson	Date	9/1/98
Program	MeanStd	Program #	1
Instructor	Philip Johnson	Language	Java

Time in Phase (min.)	Plan	**Actual**	**To Date**	**To Date %**
Planning		60	60	14
Design		40	40	10
Code		110	110	26
Compile		60	60	14
Test		120	120	29
Postmortem		30	30	7
Total	90	420	420	100

Defects Injected	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	1	1	9
Code	8	8	73
Compile	0	0	0
Test	2	2	18
Total Development	11	11	100

Defects Removed	**Actual**	**To Date**	**To Date %**
Planning	0	0	0
Design	0	0	0
Code	0	0	0
Compile	8	8	73
Test	3	3	27
Total Development	11	11	100
After Development			

Figure 2.2: *Sample Project Plan Summary form for PSP0*

duce actual, to date, and to date% values for time, defects injected, and defects removed. These values are calculated for each phase and the total process to produce numbers such as “number of defects injected in design for this project” or “total number of defects removed, to date” or “percentage of time spent in test, to date”. See Figure 2.2 for a sample form.

2.1.2 PSP0.1: Measuring Size

The main difference between PSP0 and PSP0.1 is the addition of size measurement using LOC as the basic unit. PSP0.1 introduces the Process Improvement Proposal (PIP) form to record ideas about process improvement, lessons learned, and other notes. It also adds a coding standard, which is intended to be modified by the user for specific needs, preferences, and programming languages.

These changes mean that in the planning phase the user now makes a rough estimate of the total new and changed LOC required. Additionally, since there are prior projects to reference, he or she distributes the total planned time across the individual development phases so that there is a planned amount of time for each phase. Coding should, of course, be done following the newly developed coding standard. Throughout development, the user can enter comments and ideas on the PIP form at any time. In the postmortem phase the user measures or calculates LOC in the actual completed program for eight categories: program base size, deleted, modified, added, reused, total new and changed, total LOC, and total new reused. The forms record to date totals for the LOC reused, total new and changed, total LOC, and total new reused. The user now has the data to answer such questions as, “How big did this project turn out to be compared to my initial estimate?” or “How much code was I able to reuse in this project?”

2.1.3 PSP1: Estimating Size and Time

Size estimation using the **PRO**xy-**B**ased **E**stimating (PROBE) method is the primary contribution of PSP1. Size estimation for a planned project using LOC directly is very difficult. The PROBE method recognizes the need for "some proxy that relates product size to the functions the estimator can visualize and describe." [9] For object-oriented design, objects such as Java classes make good proxies. Once the programmer has gone through a conceptual design step to determine which objects and methods to create, he or she uses historical data about method size to determine a preliminary estimated size for the current project. Then the user examines historical data, using techniques such as regression, to determine the relationship between planned size and actual size for prior projects. The programmer then uses this to refine the size estimate for the current project. He or she references historical data again to determine the relationship between planned size and actual time and between actual size and actual time in prior projects so that the planned time for the current project can be based on something more than a guess.

PSP1 changes the planning phase to include conceptual design, size estimation, and time estimation. The user now records planned values for all LOC measures and uses the size estimate to derive planned time for the project. PSP1 also changes the test phase by the introduction of a Test Report Template which allows the user to record data about tests run, data used, and results. This not only improves the testing process during development, but allows for better regression testing during future modification or reuse. The user can now answer the important question, "Given my past history and this current conceptual design, how big is my finished product likely to be and how long is its development likely to take?"

2.1.4 PSP1.1: Resource and Schedule Planning

Having a good estimate of the time it will take to complete a project still doesn't tell you *when* it will be completed. For projects requiring several days or more, PSP1.1 provides support for breaking the project down into tasks and then scheduling the tasks across available days. This is done using two new forms: the Task Planning Template and the Schedule Planning Template. The user must also calculate new process statistics on the Project Plan Summary.

PSP1.1 enhances the planning phase to utilize the Task and Schedule Planning Templates. The user also calculates planned LOC/hour, % of LOC that will be reused and % of LOC that will be new reused. During design, code, compile, and test, the user records actual values related to task completion and scheduling using the new forms. During the postmortem phase, the user calculates actual and to date LOC/hour, % reused and % new reused. Finally, the user derives the cost-performance index by dividing planned time by actual time. The user now has the data to determine things like "How fast was my development of this project compared to my historical average?" and "How much do I tend to over- or underestimate the time required for project completion" and "What percentage of my new development is reusable for future projects?".

2.1.5 PSP2: Improving Quality

With PSP2, the emphasis moves from improving estimation to improving the actual quality of the finished product. Although there are only two new forms, the Design Review Checklist and the Code Review Checklist, subjectively the PSP seems to become much more complicated at this point: it introduces two new phases (design review following design and code review following code) and adds 71 fields to the Project Plan Summary form.

In planning the user now calculates the 70% size and time prediction intervals for planned size and time. Using to date defect values and the current size estimate, the user estimates the total number of defects that will be added and removed, and their distribution across the various phases. The user also derives eight new defect-related measures such as total defects/KLOC and yield (percent of defects injected before test that are also removed before test). The development phases do not change except for the addition of design and code review phases. The textbook provides sample design and code review checklists, but the user modifies these to suit his or her personal preferences. In the postmortem phase, the user calculates all the new defect measures for the actual development of the current project as well as to date values which include prior projects. The user can now answer questions such as “How does the quality of this project seem to compare with my general quality level?” and “How many defects did I remove per hour during code review for this project?” and “How effective was my design review at detecting defects compared to my testing methods?”

2.1.6 The Higher Levels

There are two more advanced process levels described in *A Discipline for Software Engineering* [9] that were not used by students in the PSP class. PSP2.1 includes design templates and cost of quality measures. PSP3 introduces a cyclic process to help in the development of larger projects. Two new phases are added: high-level design and high-level design review. New forms include a PSP3 Design Review Checklist and an Issue Tracking Log.

2.2 Reports

If a programmer follows the standard PSP processes, he or she will hopefully be able to produce better time estimates, better quality estimates, and a higher quality product. Additionally, the required calculations produce measures that are not internally necessary to the PSP, but provide developers with useful or interesting insights into their work. Examples include the cost-performance index, percentage of code that is reused or new reused, and yield. However, some questions that require analysis of multiple projects, such as "What is my defect density trend over the past three months?", are left unanswered.

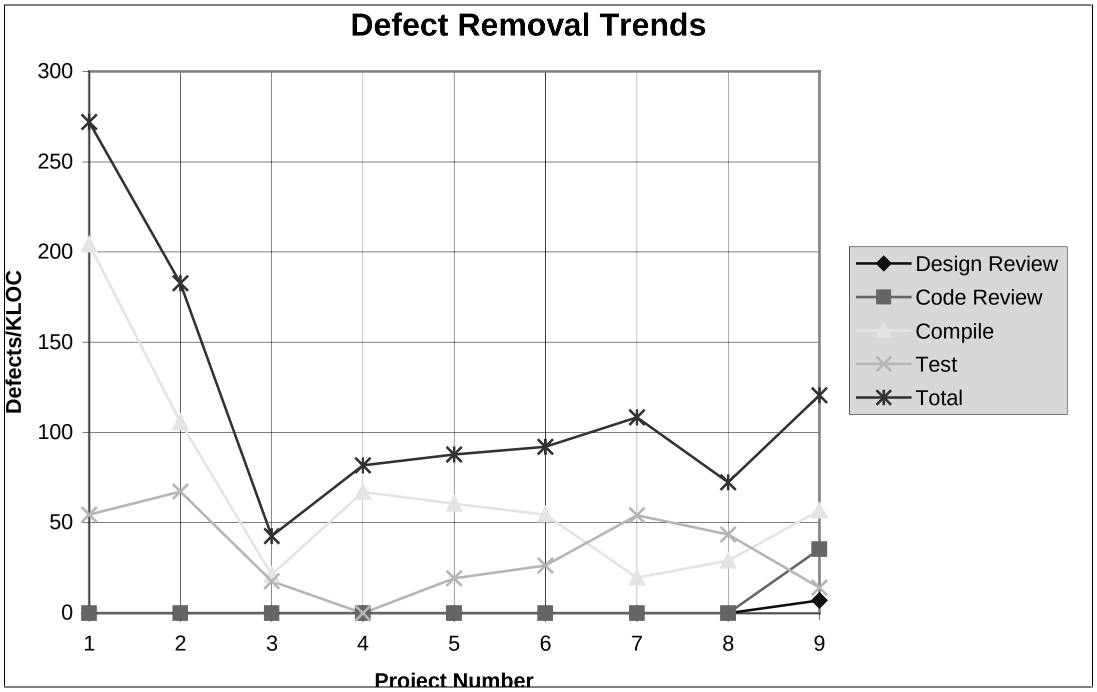


Figure 2.3: *Sample Defect Report*

The standard PSP curriculum calls for five report exercises, which outline several useful reports and provide ideas for others. The focus is primarily on defect data, but also includes estimation accuracy trends and analysis of the review phases. The reports require that multiple projects be analyzed, but not in the "to date" way used in the standard PSP processes. A sample report is shown in figure 2.3. Reports must be produced by hand using the forms from completed projects, or by using tools such as spreadsheet programs that must be created or otherwise obtained.

2.3 User Modifications of the PSP

The PSP is described in a careful and detailed way in *A Discipline for Software Engineering* [9]. However, it is not Humphrey's intention to imply that this is the ultimate personal software process for any individual developer. Although the PSP levels in the book are intended to be followed exactly in a training environment, programmers are encouraged to experiment with modifying and refining the given processes to meet their own needs. For example, some people may work in an environment where time estimation is not important. They could then modify the process to eliminate time estimation steps. In another situation, a programmer may be working with a non-object oriented language, and many need to modify the PROBE method for size estimation.

2.4 Current Status of the PSP

Currently, the PSP is being taught or has recently been taught in many academic settings, including the University of Hawaii, McGill University (Canada), University of Calgary (Canada), Carnegie Mellon University, University of Massachusetts Dartmouth, Embry-Riddle Aeronautical University, Texas Tech University, University of Utah, University of Colorado, Swinburne University (Australia), Griffith University (Australia), University of Karlsruhe (Germany), and the Politecnico di Torino (Italy). There is a PSP mailing list with about 200 subscribers. The PSP is referenced or discussed every two weeks or so on the comp.software-eng newsgroup. A wide variety of PSP forms and tools are available on-line, mostly variations of the standard PSP forms and time and defect logging tools [20]. The Carnegie Mellon Software Engineering Institute offers presentations, on-site consultations, and PSP training for instructors, managers, and students. Additional training courses are offered in industrial settings by graduates of the instructor training classes. It is hard to judge the extent of industrial adoption, but several papers are based on data from developers working in an industrial environment [6] [8] [4] [16] [22].

Chapter 3

Related Work

This chapter will discuss some of the research about the PSP, the idea of measurement dysfunction, and human error. The definitive source for PSP information is *A Discipline for Software Engineering* [9]. Chapter 2 discussed this work extensively, so it will not be covered again here.

3.1 The PSP

As Armour and Humphrey point out in their technical report on software product liability, more and more common products are controlled by computer chips [1]. As the instructions used become increasingly extensive and complex, more and more opportunities arise for software failure. This is especially important in the area of safety-critical systems, where a software error can cause harm or even death. Although preventing defects is difficult, it is easier and cheaper to prevent them than to fix problems after a product has been released. In fact, it can be up to 100 times more expensive to fix a bug than to prevent it from occurring in the first place. Software engineers need to move beyond the “build, test, and fix quality technology” to a development process that focuses first on quality and defect prevention.

In a 1996 article, Humphrey explains how the PSP is intended to work hand-in-hand with the CMM [10]. “While the CMM enables and facilitates good work, it does not guarantee it... This is where the PSP comes in, with its bottom-up approach to improvement. PSP demonstrates process improvement principles for *individual engineers* so they see how to efficiently produce quality products.” He then describes the development of the PSP. Initially, it had no levels to allow gradual introduction of the complex methods. It was so difficult to learn that, in its complete form, he was unable to get any industrial engineers to use it for actual development. Eventually he broke the PSP down into separate components that could be introduced one by one during a training course. He then reports results from some of the first PSP classes, given in both academic and industrial settings. The results are based on 104 engineers. There were more students in the classes, but their data was not used because they, “reported either incomplete or obviously incorrect results.” Results included an increase in time estimation accuracy: For assignment 1, 32.7% of the subjects estimated the time required within 20% of the correct time. By assignment 10, this number had increased to 49.0%. Reported defects fell from an average of 116.4 defects per KLOC for assignment 1 to 48.9 defects per KLOC by assignment 10.

In his master’s thesis, Dellien describes his attempt to introduce a tailored version of the PSP into an existing industrial organization [4]. To do this he used a “box-of-tools” approach to the PSP. He analyzed the PSP, broke it down into components such as measurement and quality management, compared these components with the existing processes used by the organization, and rebuilt a modified version of the PSP that addressed felt development needs but did not result in overlapping organizational processes. The he used the three steps of prestudy, education, and enactment to help software developers to begin using the PSP.

He states that using PSP in an industrial setting is different than using it in an academic setting for several reasons. First, if PSP data is made at all public, there is a temptation to alter that data, thus threatening personal integrity. Second, the PSP must somehow be made to fit in with multi-person projects and existing development processes. Third, too many forms combined with a lack of automation make training and acceptance difficult. Finally, some people may resist using the PSP because it requires a change in their work habits. In his conclusion, he observes that it is difficult to objectively evaluate whether a PSP introduction has been successful or not, even when success is measured purely as cost-effectiveness.

Sherdil and Madhavji used the PSP as the basis of their research on human-oriented improvement in the software process [21]. The first goal of their study was to measure the rate of the subjects' progress, using such variables as productivity, defect rate, and estimation error. Their second goal was to determine how much of the progress was due to task repetition (first-order learning) and how much was due to "technology injection" (second-order learning). They used the standard PSP measures to track progress, and treated the PSP size estimation method and code reviews as the technologies injected. Interestingly, they attempted to verify the subjects' PSP data by checking "... for consistency, accuracy and logical validity. Automatic tools were also used to verify the program size values. We also checked if two subjects were illegally exchanging code, but never found such an occurrence." Using this data, their study (replicated twice) showed an average progress rate of about 20%. Code reviews, introduced after project 6, helped subjects to have about 13% fewer defects over the next three projects than would have been expected had the first-order learning curve continued. Similarly, the size estimation method, introduced after project 3, reduced size estimation error by about 7% beyond what would have been expected.

Probably the most extensive study on the PSP to date was reported by Hayes and Over in 1997 [8]. It involved 298 engineers who spent more than 15,000 hours writing over 300,000 LOC and removing about 22,000 defects, during the course of 23 separate PSP training programs in both academic and industrial settings. The results reported by Hayes and Over provide very impressive evidence to support the effectiveness of the PSP. Over the projects completed, the median improvement in size estimation was by a factor of 2.5, meaning that 50% of the engineers reduced error in their size estimates by a factor of 2.5 or more. The median improvement in time estimation was by a factor of 1.75. The median reduction in overall defect density was by a factor of 1.5. This included a substantial reduction in the percentage of defects surviving to both the compile and test phases, indicating that the engineers' habits had changed in such a way that defects were caught much earlier in the development process, rather than being "tested out". Changes in productivity were very minor, and were statistically insignificant between PSP0 and PSP2.

Although the PSP classes were taught by SEI-trained instructors, the course material was not necessarily "standard" PSP, since an unspecified number of instructors taught tailored versions of the PSP. It was not stated how widely these versions of the PSP differed from "standard" PSP. In the area of design review, for example, it was clear that at least one class used design review from the first assignment on, while other instructors never introduced it at all. In another area, size measurement, there were 24 - 89 projects per assignment with no size reported, presumably because of a tailored version of the PSP or because of incomplete data. This kind of variation naturally leads to unknown effects on the data. Many instructors chose to exclude project 10, so the number of projects available per assignment varied widely. For example, when measuring defects, there were 277 projects available for assignment 1 and only 150 projects for assignment 10. Additionally, the drop-out rate was high, so that even for assignment 9 there were only 202 projects available, about 73% of the number for assignment 1. It is not clear how and when the data for project 10 or for the dropouts was used. It was specifically excluded for some results.

The instructors all had spreadsheets of differing types to perform the analysis stage calculations using data taken from the subjects' paper PSP forms. This presumably helped the instructors to obtain analysis-stage data that was more accurate than that produced by the subjects' manual PSP activities. However, the report did not mention any steps taken by the instructors to verify the accuracy of the subjects' primary PSP data. The authors appear to make the assumption that quality as measured by number of recorded defects corresponds to the actual quality of completed projects.

The authors do acknowledge some of these problems, stating, “The data collected during PSP training was designed to help engineers monitor and improve their processes, not to help in writing this report.”

Humphrey and others reported similar results in 1997 *IEEE Computer* [6]. They reported on the results of three case studies involving groups of software engineers from Advanced Information Services, Motorola Paging Products Group, and Union Switch & Signal. In these studies, quality was evaluated at least partially by such measures as acceptance test defects and use defects.

Turning to the experiences of an individual developer, Worsley reported on his impressions of the PSP after completing the 10 recommended assignment in *A Discipline for Software Engineering* [23]. Overall, the quality of his work products appeared to improve when measured as the number of discovered and recorded defects per KLOC: 62.5, 19.2, and 45.9 for the first three projects versus 5.9, 26.7, and 8.9 for the last three projects. However, this improvement was at the expense of productivity: the LOC per hour for the last three projects was less than half that of the first three projects. He states that the PSP can be difficult to learn and that it “takes a great deal of effort to follow it,” but concludes that “the knowledge and insight into your programming is very rewarding.”

A review of PSP principles and a possible future development for the PSP is covered in a series of three articles by Humphrey in CrossTalk [12] [13] [14]. He lists four criteria for successful introduction of the PSP: The engineers must have proper training, the training must be done for teams or groups, there must be strong management support, and the team members must learn how to use PSP as a team as well as individually. To meet this last requirement, he has been developing the Team Software Process (TSP). Some of its objectives are to help in team-building; to help software teams produce high-quality, on-time work; to aid managers in coaching and motivating software teams; and to help organizations in fostering CMM Level 5 behavior. The TSP will guide teams through the four phases of requirement definition, high-level design, implementation and integration/test.

Currently, it involves 23 scripts defining 173 launch and development steps, 14 forms, and three standards. Sample results are given for one group of engineers, showing improved time estimation and improved quality (measured as number of acceptance test defects).

3.2 Measurement Dysfunction

The term “measurement dysfunction” seems to have been introduced by Austin in his book *Measuring and Managing Performance in Organizations* [2]. He states that, “Dysfunction’s defining characteristic is that the actions leading to it fulfill the letter but not the spirit of stated intentions.” Measurement dysfunction describes a situation where people try, consciously or unconsciously, to change a measure used for evaluation, without trying to change the actual underlying behavior or result that is being measured.

As a classic example of measurement dysfunction, Austin cites the apocryphal Soviet boot factory, evaluated by party leaders based on the measurement of “number of boots produced”. In order to meet their quotas, factory managers produced only left boots, size 7. He reviews in more detail the case of an employment office reported by Peter Blau in 1963. The goal of the office was, of course, to find jobs for their unemployed clients. Initially, the employment office employees were evaluated primarily by the measurement of “number of interviews conducted”. They responded by focusing as much time as possible on doing interviews, and very little time in actually finding jobs for their clients, resulting in fewer clients receiving job referrals. Management then changed the evaluation measure to one comprised of eight different indicators. Employees then changed their behavior in a variety of ways to improve their standing against various indicators. These changes included destroying records of interviews that did not result in job referrals and in making referrals that represented poor job-client matches.

In both these examples of measurement dysfunction, the true performance of the organization decreased over time, even while the measurement indicators of the level of performance improved.

Austin then proceeds to divide measurements into *motivational measurements* which are “explicitly intended to affect the people who are being measured”, and *informational measurements*, which “are valued primarily for the logistical, status, and research information they convey”. Examples of using measurements for motivation include the use of sales figures for bonuses or rewarding strong performance with increased changes for promotion. Examples of using measurements for information include evaluating measurements about an existing process to design a more efficient one and measuring inventories in order to keep the optimal amount of needed supplies on hand.

He argues that the only way to ensure accurate informational measurements is to strictly segregate the two kinds of measurement based on their intended use. Unfortunately, this is difficult to do. Very often there is, intentionally or unintentionally, an actual or perceived use of an informational measure for evaluation purposes. This will almost always result in a manipulation of the measurement, making it less than useful both as a piece of information and as motivation of truly productive behavior.

3.3 Human Error

The research reported by Hayes and Over shows a high rate of error during software development, even when measured as programmer-reported defects [8]. At the level of PSP0 quality is not yet the primary focus of the PSP. At this initial level, engineers reported 94.3 defects per KLOC. Even after quality was emphasized in PSP2 through design reviews, code reviews, and various quality measures; engineers reported 25.5 defects per KLOC (most of which were reported to be removed before testing).

In research done by Panko and Halverson, subjects were asked to develop a pro forma income statement using a spreadsheet, based on a word problem [19]. The subjects made relatively few errors at the cell level (0.9% to 2.4%), but these errors tended to have a "ripple effect", so that the bottom line figures contained a large number of values (53% to 80%). The paper cites many other studies showing similar quality problems in a wide variety of areas.

These two examples help to illustrate that human error is a problem in computer-related tasks ranging from software development to spreadsheets. Errors in the PSP are similar to spreadsheet errors in the sense that a single error can end out affecting many other data fields. Additionally, spreadsheets are often used to perform analysis stage PSP calculations.

The PSP is intended to reduce the effects of human error in software development tasks. However, it is interesting to ask, "To what extent can human error affect the PSP itself?" The following chapter describes the case study I performed to explore this question.

Chapter 4

Case Study

To investigate the issue of data quality in the PSP, Dr. Johnson taught a modified version of the PSP curriculum over the course of a semester in 1996. Ten students participated in this course, completing nine projects each. (A final project for one student had so many incomplete values that I excluded it from the study.) At the end of the course I collected these projects and then examined them to uncover any errors the students may have made in filling out the PSP forms. This chapter will cover the case study in more detail, describing how Dr. Johnson modified the standard PSP curriculum, the method I used to record errors, the rules used to correct the original data, and the comparison of the original and corrected data.

4.1 The Modified PSP Curriculum

In order to address data quality problems discovered in previous semesters, Dr. Johnson made some changes to the standard PSP curriculum before teaching the course:

Increased process repetition. The standard PSP curriculum assigns ten programs during the course, in addition to several midterm and final reports. Over the course of these ten programs, students practice seven different PSP processes, which means that the development process used by the students changes for seven out of ten programs. After his initial experience teaching the PSP, Dr. Johnson came to believe that the high 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 PSP curriculum included only five PSP processes, enabling students to practice most processes at least twice before moving on. The modified curriculum also included only nine programs instead of ten, providing additional time in each assignment for data collection and analysis.

Increased process description. In his initial experience teaching the PSP, Dr. Johnson also 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 data from prior programs. To help resolve this and other similar problems, he added four additional worksheets to the standard PSP form set: (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 (this proved invaluable to me later on - it allowed me to determine whether errors in size and time estimation occurred during data transfer between projects or in the actual calculations).

Technical reviews. At the completion of each project, students divided into pairs and carried out an in-class 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. The review took about 60 minutes to complete. The students then submitted the technical review forms with the completed projects. Dr. Johnson reviewed the projects a second time for grading purposes, using the Technical Review Defect Recording Log to record any additional errors.

Tool support. Finally, Dr. Johnson provided four spreadsheets to support records of planned and actual data values. In addition, students had access to tools to count lines of code for Java programs and to compare two versions of a Java program and report lines of code added and deleted. In the textbook PSP curriculum, a LOC counting tool is assigned as one of the programs. The modified curriculum included completely new program assignments more suited to the Java language.

Dr. Johnson also emphasized data quality during 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 more importantly, how data is accumulated over the course of the semester. He dedicated lectures to collection and analysis of data periodically throughout the semester. He showed the class aggregate statistics on the entire class' performance throughout the semester. 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 my results. He did technical reviews of every assignment's PSP data. He habitually presented problems faced by one student to the entire class, either by e-mail or in person, in order that students could profit from each other's mistakes (of course, removing all identifying information from the material to avoid public embarrassment).

4.2 Data Entry

At the end of the course, Dr. Johnson gave me the PSP forms for all 90 projects, as well as all the technical review checklists and the instructor grading sheets for each project. At that time I designed and implemented a database system to store all of the PSP data from this course and to subject it to further data quality analysis (see Chapter 5). Then I entered each project into the PSP tool; recording primary data such as time, defects, and program size. I then compared the values calculated by the tool with the corresponding values calculated by the students. Each time an error was found, I reviewed the record sheets from the technical review process and the grading sheets used by the instructor to determine which reviewer first identified the error.

I did not check the Task Planning Template, the Schedule Planning Template, or the Object Size Category Worksheet for errors.

After entering the first few projects, it became harder and harder to find errors by simply comparing a value calculated by the student with one calculated by the PSP tool. This was because it was possible for students to do calculations correctly, but with incorrect data from prior projects. This produced values that differed from the ones generated by the PSP tool, but which could not be counted as errors (see section 4.2.1). This meant that I often had to hand-calculate certain values such as *Total LOC (T), to date* using the paper PSP forms for the current and previous project. For the more complicated calculations, such as linear regression for time estimation, I modified the PSP tool to allow me to update the inputs. The values defaulted to the database values stored in the tool, and then I overrode ones that varied from the ones actually used by the student.

After getting about halfway through project 4, I also realized that some of the primary data was showing inconsistencies. For example, there were overlapping time log entries and defect logs showing more fix time per phase than the time log showed total for the phase. Since it would be interesting to see any possible clues about a collection stage problem, I went back to the beginning and examined all the primary data for various problems. I recorded errors found under the general type of “Impossible Values”, and continued to do consistency checks as I entered the rest of the projects.

Whenever I found an error, I recorded the following information in another database:

- A code identifying the student responsible for the error.
- A code identifying the person who first identified the error; the instructor, another student, or myself.

- The number of the assignment in which the error occurred.
- The programming language used for the assignment. (As it turned out, all projects were done using Java.)
- The PSP process that the student was using when the error was injected.
- The PSP phase in which the student was working when the error was injected.
- A code identifying the general error type. For example:

BF	Blank Field
CI	Calculation done incorrectly

- A code identifying the specific error. For example:

BLC	Base LOC, actual: incorrect
BC	Base LOC, plan: different from Size Estimating Template

- A code identifying the error severity. For example:

0	Error has no impact on PSP data
1	Results in a single bad value, single form

- The age of the error. This represents the number of assignments since the introduction of the data field or PSP operation in which the error occurred.
- The incorrect value (where applicable).
- The value that should have been used (where applicable).

Time Recording Log						
Date	Start	Stop	Interruption Time	Delta Time	Phase	Comments
12/01/97	10:40	11:20	0	30	plan	total minutes should be 40
12/01/97	11:20	11:30	0	10	design	
12/01/97	12:30	12:50	0	20	code	
12/01/97	12:50	12:55	0	5	compile	
12/02/97	09:00	09:15	0	15	test	
12/02/97	09:15	09:25	0	10	postmortem	

Figure 4.1: Example of Time Recording Log Error

4.2.1 Error Counting Method

Upon finding an incorrect value, I recorded an error for the field, but from that point on I treated the incorrect data value as correct. For example, consider the first entry in the sample Time Recording Log shown in Figure 4.1.

Since it appears the user incorrectly subtracted *Stop* from *Start* when calculating the number of minutes spent in planning, an error would be recorded for the *Delta Time* field. But when looking at *Total Actual Minutes*, which is the sum of all the *Delta Time* fields in the Time Recording Log, on the Project Plan Summary form (not shown), 90 minutes would be treated as the correct value, even though the time spent in planning was actually 40 minutes, making the true value of *Total Actual Minutes* 100 minutes.

I do feel confident that every error that I recorded exists in the student data - I checked and rechecked, sometimes five or six times, to be absolutely sure. On the other hand, I don't believe that I uncovered all or even most of the errors present. While the PSP tool did enable me to determine the correctness or incorrectness of values generated during the analysis stage, it provided only limited insight into collection stage errors. For example, in a 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.

4.3 Data Correction

It could be the case that although users of the PSP make many errors, these errors are only "noise" and do not make a significant impact upon the trends and conclusions reached from the method. I attempted to look into this more closely by computing the measures of yield, defect density, and so forth for individual data and the aggregate results twice: once using the original data supplied by the students, and once using correct versions of the data produced by the PSP tool.

To do this, I copied the original database, which now contained all PSP values for the 89 complete projects. Using the second database, I then went again through each phase of each project, attempting to correct the primary data and then re-running the calculation steps.

As I attempted to correct the students' data, it soon became clear that their errors fell into two classes. Some errors I could easily correct with reasonable confidence, such as a mismatch between the number of defects entered in the Defect Recording Log and the total calculated for the Project Plan Summary. But in other cases, such as a blank *Phase Injected* for a defect, it was impossible to determine a correct value.

This can be illustrated with the Time Recording Log example used in Figure 4.1 to demonstrate the error counting method. It is clear that there is an error in the first entry, and it seems obvious that it occurred when calculating *Delta Time*. However, we really can't tell for sure which field was entered incorrectly. Should *Start* be 10:50, or should

Stop be 11:10, or was there a ten-minute interruption that was not recorded, or was *Delta Time* actually 40 minutes but calculated incorrectly as 30 minutes?

4.3.1 Correction Rules

Despite the obvious impossibility of doing a perfect job in correcting the student data, I formulated a set of rules that could be consistently applied in attempting to make corrections. Underlying all of these rules is the basic assumption that even though primary data may be faulty, it is probably more trustworthy than calculations performed upon it. So, for the Time Recording Log example above, the *Start*, *Stop*, and *Interruption Time* values would be considered correct, and the *Delta Time* value would be changed to 40 minutes.

These are the rules I used in data correction (some were only used once, or a few times):

- **Errors in Time Recording Log entries:** Assume 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, use the preceding and following entries and the delta time for the current entry to formulate plausible start/stop times. Generally this will mean starting the second entry where the first one stops. This rule was used 53 times.

- **Missing Time Recording Log entries:** 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*, formulate an appropriate Time Recording Log entry with fabricated date and time values. This rule was used 5 times.

- **Conflicts between Defect Recording Log and Project Plan Summary:** Assume 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. This rule was used 28 times.

- **Conflicts between Defect Recording Log and Time Recording Log:** 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, insert a Time Recording Log entry with start and stop times such that, combined with 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 defects. This rule was used 10 times.

- **Post Mortem phase used in Defect Recording Log:** If the post mortem phase is used on a Defect Recording Log entry for *Phase Removed*, increment the count in the *Defects Removed, After Development, Actual* field on the Project Plan Summary form once for each such defect. This rule was used 3 times.

- **Blank Injection Phase for Defect Recording Log:** If the *Inject* field is blank for a Defect Recording Log entry, but the *Fix Defect* field contains a phase name instead of a defect number, use the phase name to fill in the *Inject* field. This rule was used 37 times.

- **Blank** *Time in Phase (min.), Plan* **field on Project Plan Summary form:** Use the value for *Time in Phase (min.), Actual* for the same phase. This rule was used only once.

- **Conflicts in** *Program Size (LOC)* **fields on Project Plan Summary form and Size Estimating Template:** Assume that *Base, Deleted, Modified, Added, and Reused* are correct and that errors are the result of incorrect calculations. Note: this is not a truly satisfactory assumption because *Total LOC, Actual* should be the result of a measurement rather than a calculation and should therefore be relied upon. However, given correct values for *Base, Deleted, Modified, Added, and Reused*; *Total*

LOC can be calculated, whereas it is impossible to even guess at the correct values for the other fields. Unfortunately errors in the *Program Size (LOC)* fields were some of the most common errors. Combined with the importance of these fields in both size and time estimation and my inability to provide adequate corrections, estimates made with the ”corrected” data were undoubtedly severely affected. This rule was used over 60 times.

4.4 Data Comparison

After I partially corrected the project data according to the rule set, I decided 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 fully complete this assignment, 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, I selected a few fields from each of the other major sections of the Project Plan Summary, including some fields that represented fairly simple calculations but included 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.

Chapter 5

Software Tools

In order to carry out the case study, I developed two software tools: the first to automate the PSP and the second to track errors found in the PSP projects. Both tools were written using the Progress 4GL/RDBMS [3], version 6.3F01, running on SCO Unix, Release 3.2v4.2, using a character based interface. This chapter will describe the two tools; including requirements, operational scenarios, structure, and areas for improvement.

5.1 PSP Tool

5.1.1 Requirements

Obviously, the main requirement for this package was the ability to reliably automate all the PSP calculations so that I could check the original PSP data for errors. Since for this purpose I was the only user, a clear user interface and flexible functionality weren't very important. However, I also wanted to go beyond what was necessary for simple data checking and explore some ideas about a fully integrated set of PSP tools.

When I first learned the PSP, I almost immediately began creating various small tools to help me follow the PSP processes more efficiently. Some of these, such as a Java LOC counter were assignments in the PSP course itself. I created others, such as a size estimation applet, simply because I got tired of doing various processes by hand. Even though these tools relieved me of a lot of tedious work, it still seemed as though there was a tremendous amount of overhead involved in using the PSP to do actual software development. I would have a stack of PSP forms (with the correct one never on top) to the left of my keyboard, a stack of prior projects beyond that, *A Discipline for Software Engineering* (never open to the right page) for process scripts and form instructions under my elbow, and to the right all the papers needed for whatever software project I was working on. Of course, my pencil, eraser, and calculator always seemed to be *under* one of these papers whenever I needed them! On the computer, I had to continuously invoke various tools, provide input, and write down results on one of the PSP forms. All this led me to form the requirement that the new tool should completely eliminate any need for paper, including the need for the textbook for process scripts and form instructions. (Of course, the book is still needed to *learn* the PSP, but not as a day-to-day reference guide.) It should seamlessly combine all tools in such a way that the user never has to consciously invoke a tool or transfer data from one tool to another.

Closely related to this was the requirement that the new tool should provide as much guidance as possible in following process scripts. The user should not have to remember the order of the phases, the order of the forms within each phase, or which fields on a particular form are required for a particular phase. The tools should provide context-sensitive help screens and help messages. The computer should also calculate all possible values. It should provide derived values, such as *LOC per hour, Actual*, to the user in display rather than update mode. For other values, such as *Time in Phase, Total, Plan*, it should at least give the user a default number to override.

Since PSP results aren't any better than the data that goes into them, another requirement was that the tool should do as much collection of primary data (time, size, and defects) as possible. Data, such as defects, should be easy to collect so that a user is less likely to have an incomplete record of his or her work. The system should also collect data automatically, whenever possible, so that a user is less likely to have an inaccurate record of his or her work. For example, the user should not have to fill in time log entries or tell the tool how many LOC are in a certain program. However, the user should be able to override or correct any automatically collected data.

One of the most aggravating things about doing PSP by hand is the transfer of data between projects. Size and time estimation require data from as many prior projects as possible. The user must either maintain a spreadsheet of this data, or continually leaf through a stack of former projects, looking for the relevant fields. Even more data is required from the most recently completed project, in both the planning and postmortem phases. The user must find the PSP forms for this project, and then reference various values at least 45 times (for PSP2), not counting size and time estimation. Even worse is the situation where a programmer discovers an error in PSP data for an older project or decides to declare a prior project an outlier. Recalculating to date values through prior projects up to the current project is no fun at all! Therefore, a requirement for the new tool was the ability to do inter-project management. The tool should allow the user to categorize each project by PSP process, programming language, and process type; as well as allowing certain projects to be marked as outliers. Other than the PSP process, the user should be able to change any of these characteristics even after the project is complete. The tool should have the ability to recalculate all PSP data for all prior projects at any time. Furthermore, when the user is working on a new project, the tool should automatically reference relevant prior projects to provide all appropriate “carry forward” values.

Since reports make collected PSP data even more useful, the ability to do at least the reports outlined in *A Discipline for Software Engineering* was another requirement.

A final requirement was that the tool should be as flexible as possible. Since Humphrey intended that developers should modify the PSP for their own needs, users should be able to do such things as define new processes, change existing process definitions, maintain individual coding standards, or add new defect types. Users should also be able to customize the system by such actions as modifying help screens, adding or modifying table entries for programming languages, or changing menu option descriptions.

5.1.2 Operational Scenarios

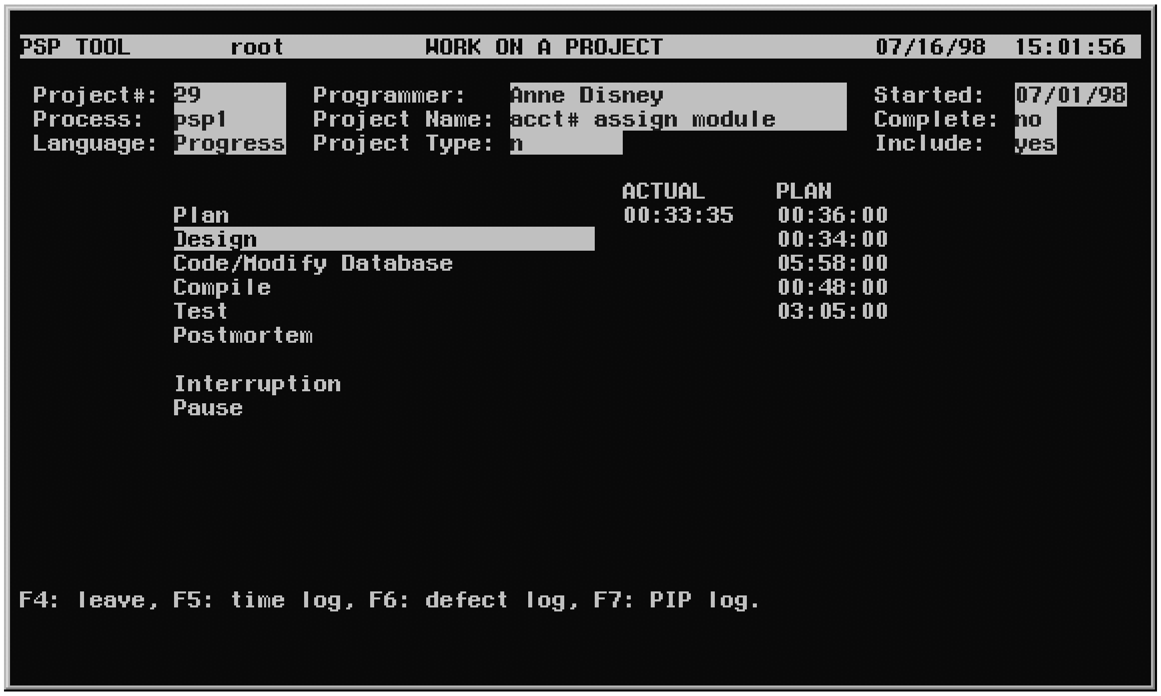
Operational scenarios are descriptions of users actually using a system. They provide a story-like rather than manual-like overview of an application. They are not intended to cover all system functions or every possible action by the the user. Instead, they provide small representations of what it would be like to actually use an application. These scenarios are examples of the way I actually used the tool while checking the student data, and illustrate how the tool could be used by future PSP users.

5.1.2.1 Scenario One: Starting a Project Using PSP1

A developer decides to start work on a new software project using PSP1. She goes to the main PSP menu and selects “Work on a Project”. The menu disappears and the header for the main project screen comes up. This header can be seen on the top part of the screen shown in Figure 5.1. The system prompts her for a project number. Since she is starting a new project, she follows the instructions in the help message and hits return on the blank field.

The system looks in her personal profile for the next project number to assign, verifies that it hasn’t already been used, and assigns it to the new project, displaying “29” on the screen. It then prompts her for the other data elements needed for project management: process, programming language, programmer, project name, project type, date started, and outlier status. Based on system information and her personal profile, the system provides defaults for process, programming language, programmer, date started, and outlier status. The developer accepts these values and fills in the project name and assigns the project a type of “N” for new. She then presses F1.

The system sees from the header data that she will be using PSP1. It finds the database record previously created to define process “PSP1”, builds a list of the phases that are involved, and displays their names on the screen. There are two additional columns for



```
PSP TOOL        root            WORK ON A PROJECT                 07/16/98  15:01:56

 Project#:  29        Programmer:   Anne Disney              Started:   07/01/98
 Process:   psp1      Project Name: acct# assign module      Complete:  no
 Language:  Progress  Project Type: n                        Include:   yes

                                                ACTUAL      PLAN
            Plan                                00:33:35    00:36:00
            Design                                          00:34:00
            Code/Modify Database                            05:58:00
            Compile                                         00:48:00
            Test                                            03:05:00
            Postmortem

            Interruption
            Pause



F4: leave, F5: time log, F6: defect log, F7: PIP log.
```

Figure 5.1: *PSP Tool: Main Project Screen*

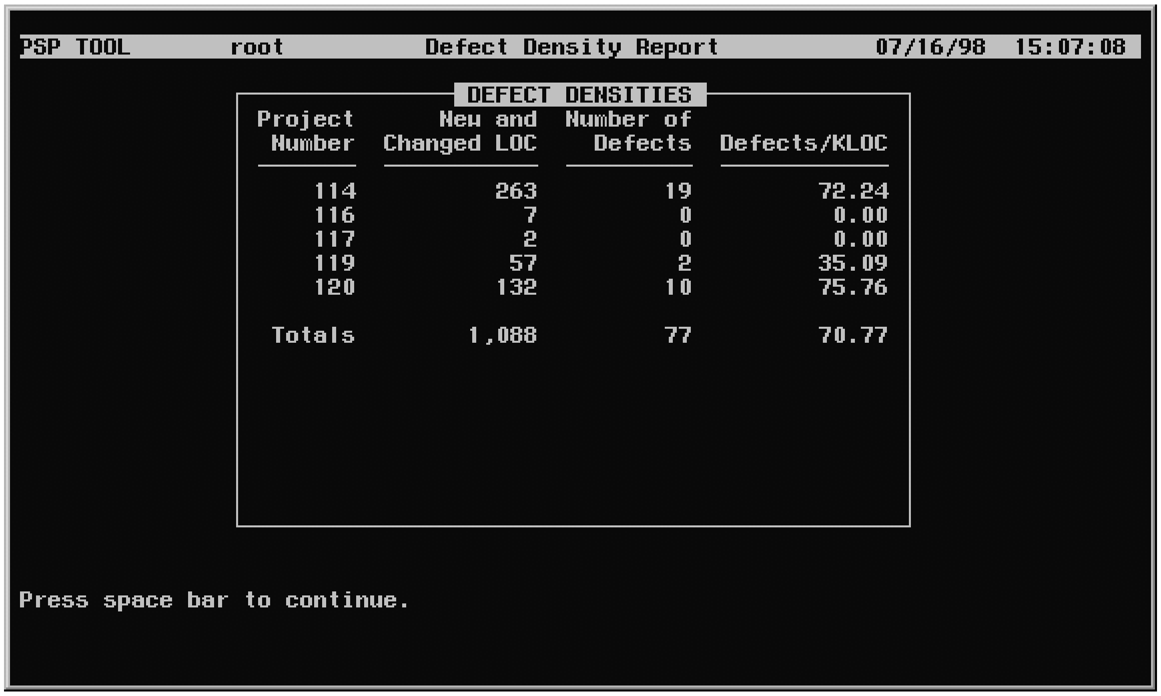
Planned time in phase and *Actual* time in phase. The developer knows that although she can arrow up and down over the phase list and select any one, the first phase listed is the phase she should use first if she plans to follow the standard process script. Therefore, she presses return on “Plan”. Immediately an asterisk appears by “Plan” on the screen, showing that this is the current phase. Under the *Actual* Time in Phase column, the amount of time spent in Planning is incremented second by second.

Immediately, a box appears titled, “Problem Description”. She types in “Client LYX requests WP interface to dictation sequences.” and presses F1. The box disappears and a new box comes up titled, “Requirements Statement”. She types in a number which references the hardcopy requirements document generated in a planning meeting the previous week, and presses F1. A text editor then fills the screen with a document already named, created, and pulled up for her, titled “Produce a Conceptual Design”. She takes about half an hour to do this work, and exits the editor. A new box appears titled “Program List”, which prompts her for the code location and the names of the programs she will be adding or modifying. The code location defaults to the one stored in her personal profile, so she just types in the name of the one program she will be modifying and the names of four new ones. After she presses F1, the system automatically counts and displays the number of lines in each program (zero for the new ones). It also makes backup copies so that after development it can determine the number of lines that were added, modified, or deleted.

Then the “Size Estimating Template” box appears, which prompts the developer for all the fields in the top part of the standard Size Estimating Template. After she fills them in and presses F1, another box appears, showing the results of all calculations done from the lower part of the standard Size Estimating Template. There is a message in the box informing her that calculations have been done, “Using projects from current language/process/type.” The “Project Plan Summary” box appears, showing default values for *Program Size (LOC), Plan*: base, deleted, modified, added, reused, and total new reused. She glances at them and presses F1. Then the time estimation box appears, with an estimate for total time and a message “From regression calculation on estimated object LOC and actual hours.” She accepts the default value of 661 minutes and presses F1. The system uses to date percentages from her most recent similar project to distribute the time across the phases, and again presents her with default values. She presses F1 again and is returned to the main project screen. The screen now displays the planned values for each phase, and “00:33:35” under *Actual* for the phase “Plan”, as shown in Figure 5.1. She moves the cursor down one line to “Design”. Invisibly, the system creates a time log entry for the planning phase.

5.1.2.2 Scenario Two: Defect Density Report

A software engineer has been conscientiously using the PSP for about six months, and wonders if the quality of his work on new projects is improving or not. He goes to the main PSP menu and selects “Reports”. The system displays the Reports Menu and he selects “Defect Densities”. The system hides the Reports Menu and displays the driver screen for the Defect Density Report. The screen prompts him for process, project type, language, date range and display type. Using his default values stored in database file, it fills in “PSP2” for process, “N” for project type (new projects), “Progress” for language, 06/12/98 and 09/04/98 for date range (first and last project completed of type “N” using PSP2), and “D” for display type. He doesn’t need to change anything, and presses F1 to proceed. Instantly, a box appears with a line for each of the 25 new projects he completed in Progress using PSP2 during the selected date range. For each project the system displays project number, new and changed LOC, number of defects, and defects per KLOC. The last line provides total values and the average number of defects per KLOC, as shown in Figure 5.2. He is happy to see that the later projects do appear to have fewer defects.



```
PSP TOOL         root            Defect Density Report            07/16/98  15:07:08

                          DEFECT DENSITIES
                 Project      New and  Number of
                  Number  Changed LOC    Defects  Defects/KLOC
                 -------  -----------  ---------  ------------

                     114          263         19         72.24
                     116            7          0          0.00
                     117            2          0          0.00
                     119           57          2         35.09
                     120          132         10         75.76

                  Totals        1,088         77         70.77

Press space bar to continue.
```

Figure 5.2: *PSP Tool: Last Page of Multi-Page Defect Density Report*

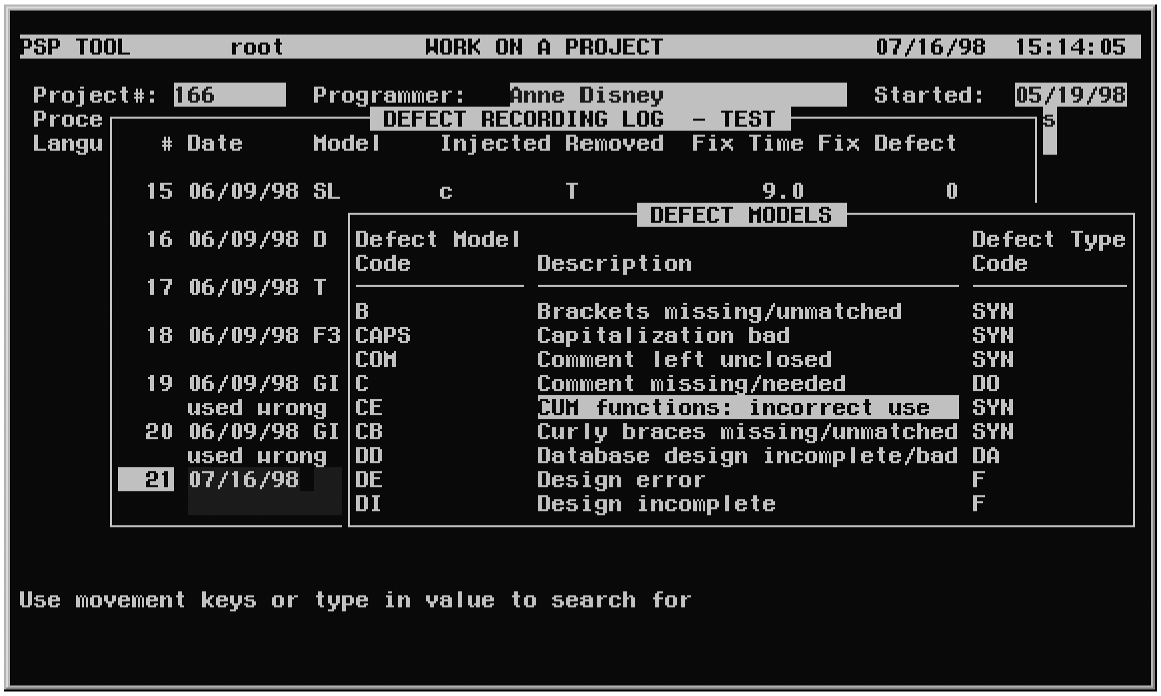
5.1.2.3 Scenario Three: Adding a Defect Model

Note: In the PSP tool, *defect types* are used as defined in *A Discipline for Software Engineering*, and refer to general defect categories such as Syntax, System, Environment, and Data. I have added a new classification, *Defect models*, which refer to specific defects within a defect type, such as “Missing semicolon” or “Confusing field label”. There are three reasons for this setup: 1) The user can record in the Defect Recording Log exactly what went wrong. This eliminates the mental overhead of deciding which defect type to use and cuts down on typing in explanatory comments. 2) It provides for a more consistent classification of defects. 3) It allows for finer grained defect reporting without extra work and without sacrificing any of the functionality required to produce the standard PSP defect reports using defect type.

A programmer notices that when recording defects he is continually using the defect model code “GSYN” for “General Syntax Error” and then using the comment line to explain “CUM function set error”. He decides that he wants to add a new defect model to the system to make his defect recording process faster. He goes to the main menu and selects the “Maintain Tables” menu option. A new menu appears. He then selects the “Defect Models” menu option. The menu disappears, and the Defect Model Code Maintenance screen appears.

Following the instructions in the help message, he decides on a new code “CE” and types it in. He then enters a description “CUM functions: incorrect use” and moves to the next field which is “Defect Type Code”. He doesn’t know what the defect type codes are, so he presses F6 and a list of all defect types appears. Using the down arrow key, he moves through the list until “Syntax” is highlighted and presses return. The defect type code “SYN” is filled in automatically, and the addition of a new defect model is complete.

The next time he is testing, he finds a defect caused by his incorrect use of a CUM function. He opens a new defect in the Defect Recording Log. The system numbers the new defect, fills in the date automatically, and places his cursor on the “Model” code field. He doesn’t remember the new code for CUM errors, so he presses F6 and is presented with a list of all defect models. He types in “CUM” and presses return. “CUM functions: used incorrectly” appears under his cursor, as shown in Figure 5.3, and he presses return again. “CE” is automatically placed in the defect model code field. No comment is necessary.



```
PSP TOOL        root            WORK ON A PROJECT               07/16/98  15:14:05

 Project#: 166         Programmer:    Anne Disney                Started:   05/19/98
 Proce              DEFECT RECORDING LOG  - TEST                                  s
 Langu     # Date      Model     Injected Removed  Fix Time Fix Defect

          15 06/09/98 SL        c        T           9.0          0
                                                 DEFECT MODELS
          16 06/09/98 D  Defect Model                                    Defect Type
                         Code          Description                       Code
          17 06/09/98 T
                         B             Brackets missing/unmatched        SYN
          18 06/09/98 F3 CAPS          Capitalization bad                SYN
                         COM           Comment left unclosed             SYN
          19 06/09/98 GI C             Comment missing/needed            DO
             used wrong  CE            CUM functions: incorrect use      SYN
          20 06/09/98 GI CB            Curly braces missing/unmatched    SYN
             used wrong  DD            Database design incomplete/bad    DA
          21 07/16/98    DE            Design error                      F
                         DI            Design incomplete                 F


Use movement keys or type in value to search for
```

Figure 5.3: *PSP Tool: Browsing for a defect model while adding a new entry to the Defect Recording Log*

5.1.2.4 Scenario Four: Correcting a Completed Project

A software developer has an extra half hour late one Friday afternoon. She doesn’t want to start anything new, so she starts going through the stack of papers that tends to accumulate on one of the back corners of her desk. She finds a slip of paper that says: “McNall lab interface, 3 hours design, 03/22”. She remembers that one Sunday back in March she did some work at home and never recorded it in her PSP database.

She turns to her screen and selects “Work on a Project” from her main PSP tool menu. The system displays the main project screen and prompts her for “Project#”. Of course she can’t remember the project number, so she presses F6 and selects the correct project from the list that appears. When the header fields appear, she changes the project status to “Incomplete” and presses F1. Following the help message on the bottom of the screen, she presses F5 and the project time log appears. She verifies that no time was recorded for March 22. Again following the help message, she presses F9 and creates a time log entry for the three hours of design work. Then she presses F4 and returns to the main project screen. She selects the postmortem phase. All the recalculations take about 1 second. She then presses the space bar as 6 boxes showing postmortem data appear. The system automatically marks the project as complete, and returns her to the main project screen, where she presses F4 twice. When the system returns her to the main PSP menu, she selects “Recalculate To-Date Values”. A message appears “Working on project number 1.” The project number quickly changes as the system recalculates the to date values for all appropriate projects using the new time measures for the changed project. 10 seconds later, the system is completely updated.

5.1.3 Structure

The PSP tool consists of about 80 programs and include files (subroutines) and a Progress database. The programs and include files add up to approximately 7000 lines of code. The database has 24 files (tables), as shown in Table 5.1, containing 237 fields (columns). Each field definition includes a default validation expression, help message, display format, field description, and screen label.

I designed the system so that certain database records (rows) must be pre-loaded for the system to function. These records all belong to files whose names begin with the prefix “s-”, and include menus, menu options, and links between certain fields and programs providing “browses” of user-defined acceptable values. Unless the user modifies the system

Table 5.1: Database Files for the PSP Tool

File Name	Description
s-browse	Ties field names to code look-up browses
s-menu	Numbers and descriptions of menus
s-option	Numbers and descriptions of menu options
c-control	Default values for individual users
c-dmodel	Defect models (general defect categories)
c-dtype	Defect types (specific defect categories)
c-form	PSP forms
c-help	Help screens by process, phase, form
c-language	Programming languages
c-otype	Object types (for size estimation)
c-phase	PSP phases
c-process	Holds ordered sets of phases for processes
c-prophase	Ties phases to processes
c-worktype	Project types
defect	Defect Recording Log entries
lesson	Lessons learned/notes
PIP	Process Improvement Proposals
prob-desc	Problem descriptions
program-set	Programs created or modified during development
project	Projects
requirements	Requirements
size-template	Size Estimating Template data
test-report	Test Report Template entries
time-log	Time Recording Log entries

through further programming, nothing should be added to or deleted from this record set. The only thing that may be modified are the names of menus and the names and numbers of menu options.

Unless a user wants to define his or her own processes completely from scratch, all files beginning with the prefix “c-” should also have a basic set of records pre-loaded before the user tries to start work. These records will define the standard PSP processes, forms, defect types, phases, etc. They will also provide a starting list of programming languages, object types, and defect types. Full facilities are provided to the user to modify these files and to add new records.

All the remaining files are records of the user’s work and PSP data. The central file to this group is “Project”. Records in the other files have a one-to-one or many-to-one relationship with records in the “Project” file. For example, there is one “requirements” record per project record and many “time-log” records.

I tried to design the system in such a way that programs rarely call other programs. Instead, database records determine which program to run next. Each menu option record contains a field for “program to run”. After writing a simple report program, a user can add it to his or her system simply by creating a new menu option record referencing the new program (using the maintenance functions already written). The records used to define PSP forms also contain a “program to run” field. To define a new process, a user builds a list of the phases that the process will contain, then indicates the sequence of forms to use in each phase. This makes it easier for someone to write new programs and add them into the system, since he or she doesn’t have to understand complicated program interactions.

5.1.4 Areas for Improvement

As with most applications, the PSP tool has several areas that could use more work. The main deficiency is that the standard PSP has not been completely implemented. The only processes that are truly complete are PSP0 and PSP0.1. PSP1 is missing an entry mechanism for the Test Report Template. This is because I was never able to devise a smooth way of switching between this form and the Defect Recording Log. PSP1.1 does not include the Task Planning Template and the Schedule Planning Template. The Design and Code Review Checklists are not available for PSP2. Finally, PSP3 has not been implemented at all.

There are also two problem areas. The first is measuring program size. At least for programs written in the Progress 4GL, this is not a simple matter of counting the number of lines in the program, since what I really want to measure is the number of statements used. This involves a tremendous amount of string parsing. I was able to implement this accurately in the Progress 4GL, but it takes a long time - about .1 seconds per line. Since this is tightly integrated with the rest of the PSP tool, it is still an improvement over other line counting methods, but the function should probably be rewritten in C++ or even a Unix script, and then called by the PSP tool. The second problem involves time recording. When a user moves too rapidly between phases, a time log entry is not recorded. I'm not sure if this is a problem with my code or the outcome of a combination of quirks in the Progress 4GL.

Of course, there are also areas that could be enhanced. It would be nice to add a table for editor types which would include pointers to programs to run to create, update, and access documents using editors other than vi! Then each user's personal profile could keep track of their preferred editor and automatically use it when needed. In another area, the *Actual* time in phase column on the main project screen should automatically refresh itself after the user manually changes the time log. Similarly, when a user changes any header data for a completed project, the system should automatically recalculate all to date values for the projects that follow it, rather than requiring the user to remember to take a second step. Finally, the programs that provide maintenance of the defect types, defect models, phases, forms, etc., do not allow the user to delete entries. The user should be able to request a deletion, then the system should check all relevant records to see if the code has been used at all. If not, the system should delete the record, otherwise it should issue a warning message to the user.

5.1.5 Personal Experiences With PSP Automation

My progress toward the goal of automating the PSP was recorded in various journal entries over the eight months after my first introduction to the PSP. Initially, I was not aware of any problems with PSP data quality; instead, I was concerned with the amount of time manual PSP required.

It began with a interest in the PSP and the insights it provided into my own software development habits:

> February 5, 1996
> I'm really enthusiastic about the PSP. ... My software development group at work has absolutely no process of any kind. I've tried using similar things before, but they take too much time and don't do much good. This one seems to be different, however. I've been using it for three or four days at work, and I think it's improving the quality of my work already. After 10 years of programming, not many bugs actually survive beyond development, but now I realize how much time I was spending killing bugs in the testing phase that could have been prevented by spending just five minutes or so more in design. Besides, "watching" myself work keeps me from getting bored, and after almost five years on the current project that is getting to be a problem.

But very soon, the time required began to trouble me:

> February 23, 1996
> Less and less convinced that the current part of the PSP [size and time esti-

mation] will be helpful in my work. Thinking back over the large software projects that my company has done for other organizations (where estimates were important), it would have been impossible to get the kind of detail that [PSP] requires from the prospective clients. In addition, even if we could have gotten it, we could not have afforded to spend the weeks it would have required to design the system to the point where all the input screens were defined for a job that we had not even won yet! That would have cost us a good deal of our profit. Besides, on every project that we've worked on, the requirements have changed so tremendously along the course of development that it would have been wasted time. Not to mention all the detailed records you have to keep for every little programming process that take hours to maintain and process. I'm disappointed, because this was getting interesting, but maybe I'll find something useful on which to build something useful to me.

March 11, 1996
I'm wondering the best way to use PSP to cover as many programming scenarios as possible. Right now the place I'm having trouble in is with [problem] fixes. It's impossible to complete even the planning phase until the [problem] is verified and you know what caused it and how to fix it. By then, I often need about two minutes programming time to fix it, but all the PSP stuff takes about 10 minutes. So is this worth it?

Despite my concerns, I continued to use PSP at work. It was helping to improve the quality of my programs, although there were setbacks at times:

March 11, 1996
I let a major bug get into our new release (using PSP). I'm upset about it - I remember working a long time on the logic, and testing it well - and yet it was so obvious looking at it a few days ago that it would not work. What

happened? Maybe just a bad day, but that's no excuse. With PSP1.1 or whatever we're on now [in the PSP class at school] there is the test report sheet, so maybe that would have helped me to catch it. I HATE it when this kind of thing happens.

But I was beginning to move into the PSP mind set:

> March 15, 1996
> Last night, my office called with a request to have a prototype for some new reports planned, coded, compiled, tested, and on the computer in Pennsylvania by 8AM EST for an important demo. My first impulse was to say "This is such a huge job, and I don't want to stay up all night, so I'll just skip the PSP stuff." But then my second thought followed automatically, "No, this will be some valuable data to collect for this type of project." I couldn't believe it!

About that time I became discouraged trying to use manual PSP in the "real world". I stopped using it at work, but started to design an automated system:

> March 20, 1996
> Yesterday I went into my office to find about 20 pages of faxes pouring out of my fax machine and all over the floor. They were all [problems] to fix or [small] improvements to provide "instant gratification" for some of our pickier users. After spending an hour on the phone clearing up questions, I had five hours to get it all done - so decided not to use PSP for this little stuff. According to my data :-) it's taking at least 15 minutes per project to do the PSP work. Felt bad not to do it, but after I get all the programs done and the database built for the PSP support, it will be much faster. It was such a relief to do the programming without filling out all those forms and timing every step.

> April 3, 1996
>
> Haven't used PSP at work since March 20. Need to develop the tools to make it easier to use. I'm just sick of filling out all those forms. As soon as school is out ...

Two months after starting to use the PSP, I became aware for the first time of problems with data quality, and the ambiguities of defect recording:

> April 12, 1996
>
> According to my PSP data, I have one design error in about two months of programming. I've mostly been doing spot improvements or error fixing, but I must not be recording design errors. How do you know when you find one? Sometimes I might change my mind about how to do something in the middle of coding, or see a better way to do it, or sometimes the requirements change in the middle of coding - guess that's what I should count. But those aren't really defects. And how can you possibly inject a defect in the planning phase?

I began to get more serious about my automation project:

> Friday, April 26, 1996
>
> Worked some more on the PSP project. Going well so far, have the menu structure, on-line help and browse functions, and started the database definition. There are a lot of fields, but the structure is obvious. Getting the defect log and time log going are the most important things - most of the other data can be figured out later. Having the usual difficulties deciding whether to store calculated values, and if so, which ones.

But it took another four months to really be usable. At that time I was able to start observing differences between manual and automated PSP:

August 5, 1996

PSP0 is basically done! Just need to add the recalculate and timer modification features.

I'm much better about recording defects with the computer than with paper. However, I'm not recording every one (some are tricky) and I'm not always thinking back to the correct injection point for the defect.

The postmortem is sort of anticlimactic now. I remember when I was using the paper and calculating it all out that it used to be kind of exciting to see where all my defects came from and the percent of time in each phase. Now it's hardly interesting. Maybe because I've done higher PSP levels since then, or maybe this is just a feature of the automation.

I began to realize that automation alone would not solve all collection problems:

August 26, 1996

The most frustrating part of PSP at the moment is in keeping track of defects. After just a day or two of using automated PSP, it became second nature to flip over to the first [virtual terminal screen] and enter the defect before starting to fix it. The problems are:

a. deciding where it was injected. My usual impulse is to pick "coding", but I need to think back to design, or most often, a bug injected not because of a design error, but because of a sketchy design that left out the area that caused the bug.

b. deciding whether a problem counts as a defect or not. For example, what if when testing a screen entry program, I don't like one of the labels (but no typo or mistake). Is that a defect? What if I was not the one that defined the table (includes label definition)? What if I defined the table as part of another project than my current one?

> September 3, 1996
> Yes, it seems that the real problem with [my] PSP right now is not the automation, but correctly tracking the defects. Another example: say my design is sloppy, and I'm in the coding phase. I have to redesign, tear out a little code, and implement the new design. How do I measure the length of time that defect takes to remove?

Eventually, things began to look up, and I committed myself to making automated PSP an integral part of my work habits:

> September 10, 1996
> Now that PSP0.1 is done (except for PIP), it is more exciting to use PSP. Things have settled down so that the figures I see are correct, etc. And the process is much less intrusive, either because I'm getting used to it or because it really is smoother to use.

Currently, I use the PSP tool for most work-related projects, and have 121 of these projects on file. Most are done using Progress, but I've also used the PSP tool to record data for SPlus projects. Processes range from PSP0 to PSP1.1, as well as 14 projects done using a non-standard process used to test the tool's flexibility.

5.2 Error Tracking Tool

5.2.1 Requirements

From the beginning, I viewed the error tracking tool as a special-purpose, single-user application; with the special purpose of recording errors found while analyzing PSP data, and the single user as myself. Therefore, the requirements were rather simple.

The primary requirement was that the tool should have an appropriate database structure for recording data about the PSP errors and should provide an entry screen that would allow me to record the errors quickly and accurately. If a field on the entry screen corresponded to a list of acceptable values, then the tool should allow me to enter only one of those values.

As I found errors, I wanted to keep track of 11 pieces of data for each one; such as person who injected, person who discovered, assignment, process, phase, form, etc. (This does not include three data elements that did not apply to every error: incorrect value, correct value, and comment lines). 9 of the 11 fields would contain codes that would represent items from lists of acceptable values. Some of the lists, such as severity levels, would contain a fixed set of values. But others, such as defect types, would probably grow as I worked through the case study. Therefore, another requirement was that the lists of acceptable values should have maintenance programs available for adding and modifying entries, and that each list should appear in selection mode after the user pressed F6 from the corresponding field on the error entry screen.

A final requirement was that the tool should allow me to analyze the error data after it had been entered. It should do this by creating various reports. Since these were to be special-purpose reports created for a single user, there was no need for fancy output formats or drivers with long lists of options.

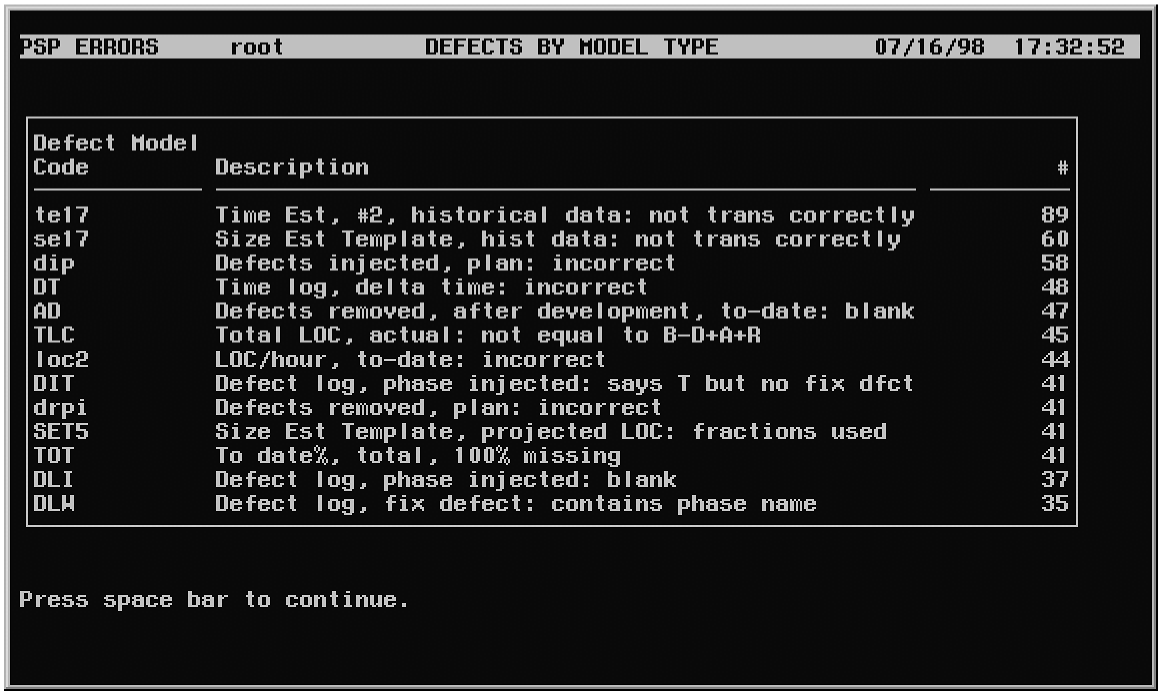
After implementing this tool and entering about half the projects, I realized that I needed to keep track of estimated LOC, actual LOC, and actual minutes as they were *originally* recorded on the student projects. This was necessary to allow me to efficiently duplicate time and size estimation and LOC calculation using incorrect data. Therefore, this became the main requirement for a small addition to the original application.

Table 5.2: Database Files for the Error Tracking Tool

File Name	Description
s-browse	Ties field names to code look-up browses
s-menu	Numbers and descriptions of menus
s-option	Numbers and descriptions of menu options
c-dmodel	Defect models (general defect categories)
c-dtype	Defect types (specific defect categories)
c-form	PSP forms
c-language	Programming languages
c-person	People (instructor, students, myself)
c-phase	PSP phases
c-process	PSP processes
c-severity	Defect severity levels (impact of errors on other data
c-worktype	Project types
defect	Errors found in student PSP data
hist	Summary of project data by author/assignment (Used to keep key data on-line so that when doing calculations by hand I didn't have to look through stacks of prior projects)

5.2.2 Structure

The error tracking tool consists of about 60 programs and include files (subroutines) and a Progress database. The programs and include files add up to approximately 2000 lines of code. The database has 14 files (tables), as shown in Table 5.2, containing 49 fields (columns). Each field definition includes a default validation expression, help message, display format, field description, and screen label. Besides providing tracking of errors and historical data, it generates 12 reports. A sample report is shown in Figure 5.4. This report shows specific errors types, sorted by the number of times each error was found during the case study.



```
PSP ERRORS      root          DEFECTS BY MODEL TYPE            07/16/98  17:32:52


 Defect Model
 Code          Description                                                   #
 ____________  ____________________________________________________   __________

 te17          Time Est, #2, historical data: not trans correctly           89
 se17          Size Est Template, hist data: not trans correctly            60
 dip           Defects injected, plan: incorrect                            58
 DT            Time log, delta time: incorrect                              48
 AD            Defects removed, after development, to-date: blank           47
 TLC           Total LOC, actual: not equal to B-D+A+R                      45
 loc2          LOC/hour, to-date: incorrect                                 44
 DIT           Defect log, phase injected: says T but no fix dfct           41
 drpi          Defects removed, plan: incorrect                             41
 SET5          Size Est Template, projected LOC: fractions used             41
 TOT           To date%, total, 100% missing                                41
 DLI           Defect log, phase injected: blank                            37
 DLW           Defect log, fix defect: contains phase name                  35


Press space bar to continue.
```

Figure 5.4: *Error Tracking Tool: Sample Report*

Chapter 6

Results

6.1 Summary Results

Despite modifications to the PSP curriculum to increase data quality, my analyses yielded 1539 data errors. In this chapter I will report on the types of errors found, their severity, their age, the manner in which they were detected, collection stage errors, and error impact. In the next chapter I will present my interpretation of these results and the light they shed on the current and future practice of the PSP.

6.1.1 Error Types

I found that the errors naturally fell into one of seven general types. They are listed below 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. Figure 6.1 provides a summary of these findings.

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.

Inter-Project Transfer Error. (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 in these cases 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 it was obvious when incorrect values originated in the transfer.

Entry Error. (142 errors, 9%). This error type applied when the student clearly did not understand the purpose of a field or used an incorrect method in selecting data. Examples include filling in the *Fix Defect* field in the Defect Recording Log with a phase name, 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.

Intra-Project Transfer Error. (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 being 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.

Sequence Error. (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.

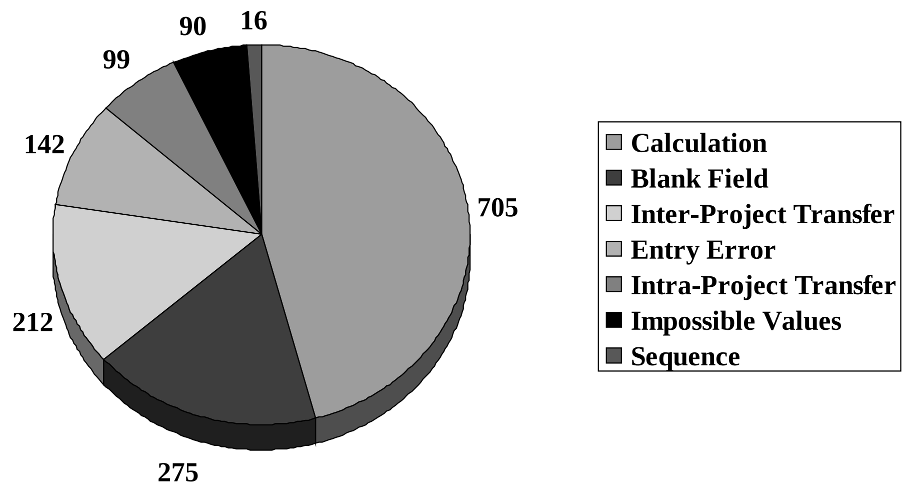


Figure 6.1: *Errors by Type*

6.1.2 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, I classified the errors into one of five "severity" levels. These levels are presented below in increasing order of ripple effect. As before, I am including the total number of errors found for a given severity level and its percentage of the total. Figure 6.2 presents a summary of these findings.

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. This will produce incorrect values not only for the current Project Plan Summary form, but for the *Defects Injected, To Date* fields for an unknown number of future Project Plan Summary forms.

6.1.3 Age of Errors

In any learning situation, a certain number of errors are to be expected. I hypothesized that perhaps the errors I 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.

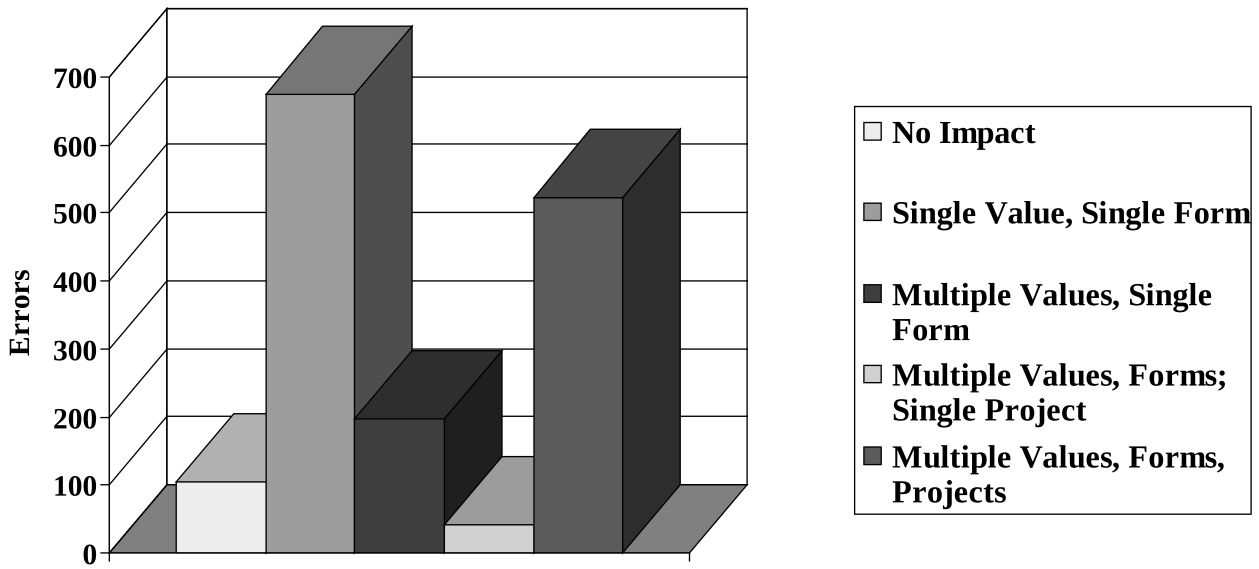


Figure 6.2: *Errors by Severity Level*

To evaluate this hypothesis, I 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 one could 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 times for the Time Recording Log entries was introduced in the first project. If a student made an error in this field during the first project I recorded a error with an age of zero. If a similar error was made during the second project I recorded an error with the age of one. By the ninth project this type of error would have an age of eight.

I first analyzed the errors to determine the average error age in each project. Table 6.1 shows the average age for all errors in each project.

I then filtered out the 309 errors with an age of zero. This eliminated errors that could have been the result of students being introduced to new fields and/or PSP operations for the first time. Table 6.2 shows the resulting data.

Table 6.1: Average Error Age by Project - All Errors

Project #	PSP Process	# of Errors	Average Age
1	PSP0a	51	0.00
2	PSP0.1a	59	0.73
3	PSP0.1a	63	1.76
4	PSP1a	150	1.27
5	PSP1a	165	2.27
6	PSP1a	186	3.30
7	PSP1.1a	160	3.26
8	PSP2a	351	3.04
9	PSP2a	354	3.84

When combining the 1539 errors from all projects, the average error age was 2.78 projects. When only the 1230 errors with an age greater than zero were included, the average error age rose to 3.48 projects.

6.1.4 Error Detection Methods

In this case study, there were three ways an error could be detected: by another student during technical review, by the instructor during the grading/evaluation process, or through the use of the PSP data entry tool. As shown in Table 6.3, students were made aware of about 5% of the mistakes in their completed projects during the course of the class.

6.2 Analysis Stage Errors

Our two stage model of PSP data quality, illustrated in Figure 1.1, indicates that errors can be introduced during either collection or analysis. Most of the errors that I found occurred during PSP analysis activities, with 700 errors occurring in the plan phase and 561 errors

Table 6.2: Average Error Age by Project Where Error Age is Greater Than Zero

Project #	PSP Process	# of Errors	Average Age
1	PSP0a	0	NA
2	PSP0.1a	43	1.00
3	PSP0.1a	63	1.76
4	PSP1a	70	2.71
5	PSP1a	165	2.27
6	PSP1a	186	3.30
7	PSP1.1a	135	3.86
8	PSP2a	214	4.99
9	PSP2a	354	3.84

Table 6.3: Errors by Detection Method

Detection Method	#	%
Grading/Evaluation (instructor)	32	2.08
Technical Review (students)	40	2.60
PSP Tool	1467	95.32

in the postmortem phase. Some of the errors occurring in other phases, such as errors in *Delta Time* calculations, were also analysis errors.

6.2.1 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. An error 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. Table 6.4 shows the four most common errors of this type.

Table 6.4: Most Frequently Occurring Persistent Errors

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

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. Specifically, many times instead of transferring *Total New and Changed (N)* (Plan or Actual), students 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 on the form the two values are vertically adjacent. It would be particularly easy to make this mistake with the Actual values because the fields are separated by one column of data from the labels. Additionally, it appeared that students made spreadsheets or ”cheatsheets” of these fields to avoid thumbing through the entire stack of completed projects every time a time or size estimation was needed for a new project. I infer this because the same incorrect value for a particular project would be transferred for time and/or size estimation in every subsequent project.

Similar factors surrounded 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, 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 48 errors in calculating *Delta Time* in the Time Recording Log were also 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. Second, 34 (71%) were in error by amounts that indicated small errors in simple arithmetic, as seen in Table 6.5. Third, the distribution of this error across projects is as shown in Table 6.6. 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.

Table 6.5: Top Four Delta Time Errors

Size of Error in Minutes	Number of Occurrences
60	16
10	9
5	5
120	4

Table 6.6: Delta Time Errors by Project

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

6.3 Collection Stage Errors

Analysis stage errors were relatively easy to find 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, I found both direct and indirect evidence for collection errors during the case study.

6.3.1 Direct Collection Error Evidence

Direct evidence of collection problems appeared in the 90 errors classified as “Impossible Values”. I grouped 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.

6.3.2 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). My analysis of the PSP data for that same project yielded 27 errors.

Finally, Dr. Johnson 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 I examined.

6.3.3 Overall Error Rate

1539 errors were found. Such a large number suggests either poor instruction or a class with a high percentage of poor students. Actually, though, there are a very large number of

Table 6.7: Data values present in the PSP

Process	Approx. Fields	Projects	Total Values
PSP0	200	10	2000
PSP0.1	220	20	4400
PSP1.0	329	20	6580
PSP1.1	437	20	8740
PSP2.0	528	19	10,032
	Total	**89**	**31,752**

data fields both for the students to fill in and for the instructor to check later. For example, a Time Recording Log entry contains six required fields. If a class of 10 students each have 10 time recording log entries per project, there are 600 data values per project related to the Time Recording Log alone - one of the most simple PSP forms.

Following this approach, Dr. Johnson has arrived at an estimate of almost 32,000 data values to be checked by hand for this single case study, as illustrated in Table 6.7. The 1539 data errors uncovered during this study represents only 4.8% of the total possible, which means that the Dr. Johnson obtained over 95% correctness (at least with respect to analysis-stage data quality).

6.4 Effects of Data Correction

To find out whether the 1539 data errors were simply “noise” that did not substantially impact upon the overall analysis results, I recalculated some of the major metrics using the (partially) corrected data. When I compared the original and corrected data, I found substantial differences in such measures as the Cost-Performance Index (CPI: planned time-to-date/actual time-to-date) and Yield (percentage of defects injected before first compile that were also removed before first compile), as shown in Figure 6.3 and Figure 6.4. [1]

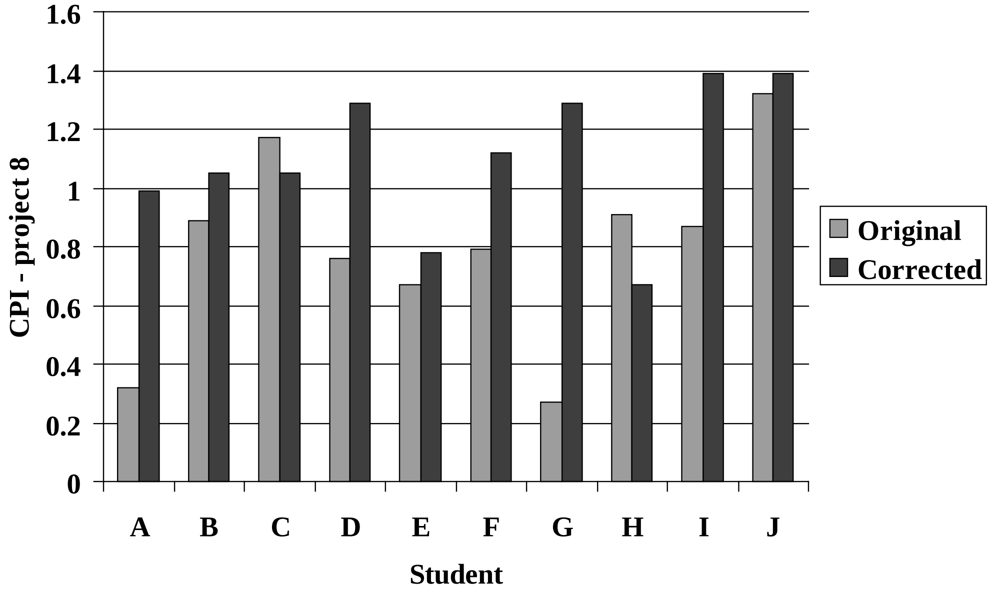


Figure 6.3: Effect of Correction on Cost-Performance Index (CPI)

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 to 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

[1] In comparing the original and corrected data, there were significant differences ($p<.05$) for CPI and Yield, when using the Wilcoxon Signed Rank Test [5], a non-parametric test of significance which does not make any assumptions regarding the underlying distribution of the data.

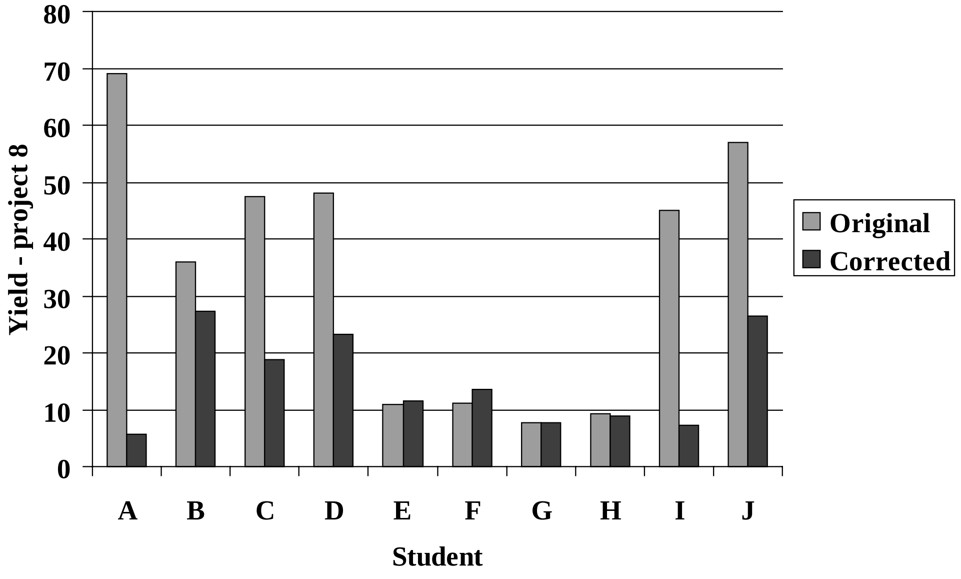


Figure 6.4: Effect of Correction on Yield

values, indicating that subjects were removing a far smaller proportion of defects from their programs prior to compiling than indicated by the Yield measurement.

6.5 Summary

In conclusion, 1539 data errors were found. Almost half were calculation errors; there were also substantial numbers of blank field and transfer errors. About one-third of the errors were of the most severe kind: errors that produced multiple bad values on multiple forms for multiple future projects. Analysis of the age of errors showed that by the ninth project, the average age of errors not related to learning a new process was about three and a half projects. Most errors were not found by instructor review or in-class technical review, but by the use of the PSP tool. Although most errors found were analysis-stage errors, there was also direct and indirect evidence of a collection stage problem. When comparing original and corrected data, there was a substantial difference for certain measures such as CPI and yield. The next chapter will cover my interpretation of these results.

Chapter 7

Discussion

The 1539 errors found in this case study give at least partial support to my hypotheses that

1. The PSP suffers from a collection data quality problem.
2. Manual PSP suffers from an analysis data quality problem.

In this chapter I will evaluate the collection and analysis data quality problems, discuss some implications of these problems in the evaluation of existing PSP data, and outline possible directions for future research.

7.1 Further refinement of the PSP model

The next few sections will cover the causes and prevention of collection and analysis stage errors. To summarize the concepts covered in previous chapters, I expand the model shown in Figure 1.1 to outline the nature of the errors associated with each stage and their correction methods:

- **Collection stage errors:** Occur in such fields as Time Recording Log start/stop times, all Defect Recording Log fields, and program size measures.
  - **Detectable:** Errors that are clearly present, despite being in primary data. Examples include PSP phases without Time Recording Log entries or missing dates in the Defect Recording Log. There are two types of detectable errors:

        * Correct value is clear or unimportant. When attempting to fix these errors, the value is not significant (e.g. a Defect Recording Log date) or can be reliably deduced from other data (e.g. a missing Time Recording Log date for a phase that has a corresponding entry with a date in the Defect Recording Log).
        * Correct value is important but unclear. These errors can never be corrected with complete confidence. An attempt can be made to fix them using a set of rules to provide consistency.
    - **Hidden:** Errors that could only have been detected by direct observation of the person using PSP and subsequent comparison with the recorded data. These errors can be made either intentionally or unintentionally.
        * Missing values. Primarily missing Time and Defect Recording Log entries.
        * Incorrect data. Data values that exist but do not reflect what actually happened.

- **Analysis stage errors:** Given reliable primary data, these can be reliably fixed by re-doing the appropriate calculations or PSP operations.

Hidden errors, which are of special interest, could have several sources:

- Missing data values of which the programmer is not aware. This includes defects not recorded because the programmer became completely absorbed in fixing a tricky bug.

- Inaccurate data values of which the programmer is not aware. For example, a faulty *Start* value in a Time Recording Log entry.

- Data values created in an attempt to recover lost information or information not recorded due to preoccupation with programming tasks, etc. For example, if a programmer is coding and is interrupted by a phone call, but forgets to time the interruption, he or she will have to guess the number of minutes to enter in the *Interruption Time* column on the Time Recording Log.

- Willful recording (or non-recording) of inaccurate data due to outside pressures such as an employer performing evaluations based on a PSP measure or a class that requires the use of PSP from a person not committed to using PSP.

7.2 The Collection Problem

In the two-stage model of the PSP illustrated in Figure 1.1, the collection stage starts with actual work and ends with records of work. Problems in this stage occur when the records of work (size, time, and defect data,) do not accurately reflect the actual work done. As stated in Section 6.3.1, 90 errors were found from the collection stage. These 90 errors lend at least some support to the hypothesis that PSP suffers from a collection data quality problem. However, as outlined in Section 6.3.2, there are indications that the actual number of collection stage errors was much higher. Analysis stage errors are relatively easy to find, since the analysis steps can be duplicated and the resulting data sets compared. The collection stage cannot be duplicated, so unless there are internal data conflicts, there is no reliable way to determine if a student padded time log entries, failed to record defects, or incorrectly measured program size.

Humphrey does acknowledge that there can be problems in this area:

> Data gathering can be time consuming and tedious. To be consistently effective in gathering data, you must be convinced of its value to you. If you do not intend to use the data, the odds are you will either not gather them or the data you gather will be incomplete or inaccurate. This is especially true for the PSP... The principal issue is whether the data you gather are for your personal use or for someone else's. If you are gathering data for someone who

sets your pay or evaluates your work, you will likely be careful to show good results. If it is for your personal use, however, you can be more objective (p. 226, 227) [9].

7.2.1 Causes

One possible cause of the collection problem is measurement dysfunction. Chapter 2 discusses this concept in more depth, but briefly, measurement dysfunction in software development is a situation in which organizational forces lead to the conscious or unconscious skewing of data to support the trends desired by management, even when the true trend is the opposite of that portrayed by the data. As a simple example, using the PSP should lead to lower defect density over time. It is easy for a programmer to achieve such an effect: simply record fewer and fewer defects in the Defect Recording Log.

The problem with teaching PSP in a classroom setting is that any grades given must be based, at least in part, on the completed PSP forms. It is difficult for the instructor to communicate convincingly to the students that the actual values recorded (if the result of correctly doing the PSP) do not impact their grades. Students can have trouble understanding, for example, that the instructor is evaluating their Defect Recording Logs based on completeness and correctness, but *not* on the number of defects injected while programming. Similarly, with LOC per hour, students may not understand that the instructor is looking for a correct computation rather than high productivity or a tendency for improvement over time. Therefore, a situation develops which is ripe for measurement dysfunction. Consciously or unconsciously, students feel pressure to improve the accuracy of their estimates and the quality of their programs. By manipulating data gathered in the collection stage, it is very easy to change the outcome of the derived measures such as LOC per hour, yield, and the cost-performance index.

Another possible cause of problems in the collection stage is pressure for time. Given that defect data collection is quite time consuming, and given that students are under significant time pressures later in the semester, it is possible that observed downward trends in PSP defect data could be due more to increasing demands upon student time than to improvements in their development skill.

Developers using the PSP collect time and defect data during the development phases of design, code, compile, and test. It can be distracting to do the work involved in these phases while simultaneously recording measures about the work. This can lead to mistakes in recording or can even cause developers to forget to record data altogether, if they become so absorbed in development that they forget that they're doing PSP at the same time. (In a worst-case scenario, a person could be so distracted by recording defects during coding that he or she could actually inject more defects than would have occurred without the PSP.) Even if a developer becomes aware that he or she has forgotten to record, say, a phone call interruption, just estimating the interruption at a later time is a source of error in itself. Errors caused by distraction could especially affect the accuracy of defect fix times. Often after fixing a defect, a programmer has no idea whether the fix took 15 or 35 minutes, and the PSP forms do not facilitate the collection of this piece of data.

Simple laziness, even fitful periods of laziness, is also a source of collection error. It is very easy to skip the recording of "little" defects or typos, especially when fixing a defect takes less time than recording it. Of course, this is closely related to a person's motivation for doing the PSP at all. If students are recording time and defects simply to get forms filled out so that they can get a good grade in a class, the completeness of the data means very little to them. Undeniably, there are times, such as when a defect is injected while fixing another defect, when defect recording is irritating to anyone. However, an internally motivated programmer is much more likely to take the small extra steps that produce accurate work measures.

Finally, there are plenty of ways to make "stupid mistakes", even when collecting data as simple as time and defects. For example, a user could write down 8:20 for planning start time when he really meant 9:20, give defects the wrong types or phases, or mix up lines added and lines deleted when copying results from a line-counting program to the PSP form.

7.2.2 Prevention

If a programmer's PSP data is used, or even viewed, by any other person, it is possible for measurement dysfunction to exist. Therefore, to prevent collection errors, any such situation should be examined to see if measurement dysfunction is occurring. If the person with access to a programmer's PSP data is an instructor or manager, the problems should be obvious, since such a person has the ability to create negative consequences for "bad" PSP measures, or to reward "good" ones. However, even peers can exert pressure. Because PSP makes personal measures such as LOC/hour very visible to peers, unspoken competition might develop among co-workers, or a student might have bad feelings about being a "worse" programmer than others in the class. In both these situations, a person could feel tempted to record fewer defects or less time for a project, therefore improving certain quality or productivity measures.

Well-designed automation would probably also reduce the number of collection errors. A good application could help to prevent many "stupid" mistakes by automatically filling in values such as start and stop times. The user would still have to indicate that he or she was leaving a particular phase, but the actual time stamp and delta time could be created by the computer. The application could also provide timing for defect fixes, which would preventing collection errors generated by requiring the user to guess fix times. It could automatically measure programs used in a project, calculating LOC added, modified, and deleted; and store the numbers for future computations in the analysis stage.

An automated version of the PSP could make doing PSP much simpler. This would free the user from some of the inherent distraction involved in using the PSP, leaving more mental room to remember to take the necessary steps for data collection. It would also allow the user to remain physically focused on the terminal and keyboard, which could particularly help in defect recording. All unwanted hand, arm, body, and eye movements away from the screen to the desktop could be eliminated; defects could be entered much faster. This might make a person more inclined to record a defect encountered during compile or test.

Finally, to prevent collection errors, PSP users must truly desire to improve their work. Even with no measurement dysfunction and a smooth-functioning automated PSP application, a person can still be sloppy about data collection. Motivation, therefore, is highly important. However, producing and maintaining a positive, meticulous mind set is a challenge that each programmer must address individually. Humphrey suggests that skilled coaches can help in this area:

> Without motivation, professionals do not excel; with motivation, they will tackle and surmount unbelievable challenges. While there are many keys to motivation, the coach's first task is to find these keys and devise the mix of goals, rewards, and demands that will achieve consistently high performance.

7.3 The Analysis Problem

In the two-stage model of the PSP illustrated in Figure 1.1, the analysis stage starts with records of work and ends with analyzed work. Problems in this stage occur when PSP users make any kind of errors in this analysis, whether incorrectly performing computations, providing the wrong data for computations, or choosing the wrong statistical methods. As outlined in Section 6.2, 1479 of the errors found in the case study were committed during the analysis stage. This lends strong support to the hypothesis that manual PSP suffers from an analysis data quality problem. The support is only strengthened by the fact that these 1479 errors do not include the thousands of fields with incorrect values that were the result of subjects performing subsequent analysis steps using this flawed data. As shown in Section 6.4, steps taken to correct analysis stage errors created a second set of data that included substantial changes for some significant measures, demonstrating that the analysis errors found were not merely “noise”.

7.3.1 Causes

Why did the subjects make so many mistakes? Part of the problem lies in the nature of manual PSP. To begin with there are a lot of forms. PSP0 starts out with four scripts and four forms. PSP1 has five scripts and eight forms. By PSP2 there are five scripts, ten forms, and two checklists. Seven of these forms and checklists are likely to extend to multiple pages for even a medium size project. Not only are there are a lot of forms, there are a lot of fields on those forms. It isn’t possible to give an exact number of fields per process, since many of the forms, such as the Defect Recording Log, have a variable number of entries. However, Dr. Johnson estimates that the average number of fields per project probably ranges from about 200 for PSP0 to at least 530 for PSP2. There are 177 fields on the PSP2 Project Plan Summary form alone.

Just having a lot of forms with a lot of fields shouldn’t make things overwhelmingly complicated. However, there are other factors involved. First, on these numerous and complex forms, not all fields are applicable to the current phase – whatever phase that happens to be. It is easy to get confused about what form and field you are supposed to be filling in now and what should be saved for a later phase. Second, there are data dependencies between the forms for a single project which involve a constant transferring of data from paper to paper. Almost every form has data that must be summarized and sent to another form, or must itself have data from another form in order for the user to complete the calculations. Every one of these transfers is an opportunity for an error to be made, either by transferring the wrong number or by transferring the right number to the wrong place. Third, there are many calculations and operations that involve prior projects. Just locating the correct project, form, and field can be frustrating and time-consuming, and there are the same opportunities for transfer errors. Particularly for size and time estimation, the user must leaf through a pile of old projects, or rely on a possibly inaccurate list of the pertinent values such as planned and actual size and time. At best, this list is yet another form to maintain. Additionally, when learning the PSP, all the scripts and the set of forms change with each new process. Therefore, as the user learns the PSP, familiar forms and scripts are constantly changing.

Another factor contributing to analysis error is the textbook [9] itself. Admittedly, the material it covers is both complex and extensive, but in some areas the instructions are not very clear. The main problem, however, is not the clarity of the instructions, but their location(s). For a single form, a PSP user might have to locate three or more references in the book. For example, the Size Estimating Template is introduced on page 120 with a sample form showing a sample project. It is discussed over the following four pages. The full instructions are not given until page 684 (Appendix C, PSP Process Elements). A sample form is shown there, but no examples - all the fields are blank. In the instructions, references are made to Appendix A, Table A27 (no page number given) for the procedure to calculate regression parameters, and to Appendix A, Table A29 (no page number given) for the procedure to calculate the prediction interval. There are also some notes about size estimation on page 621 (Appendix C, PSP Process Contents). Further instructions about size estimation appear on page 679-80 in the PROBE estimating script, phases 4-8. However, there is no information about how to combine the steps in this script with the seemingly overlapping set of instructions on page 684.

Furthermore, instructions for new fields appear at inconsistent places in the text. The formula for *Cost Performance Index* appears in Appendix C, Process Contents; while the formulas for the *Cost-of-Quality* fields appear in the text itself, in Chapter 9. An example worked out for the Size Estimating Template appears in Chapter 5, but the example for the PSP1 Project Plan Summary form is in Appendix C, Process Contents. Instructions for carrying out time estimation are given in chapter 6, but there is no index entry for “time, estimation”. The PSP user must look under “estimating, development time”. All this makes it difficult for a developer to find answers, or to feel confident that he or she has all the pertinent information even after finding a useful reference.

The actual computations are another source of analysis errors. Fortunately, the book does have a good 80-page appendix covering such subjects as statistical distributions, numerical integration, tests for normality, linear regression, multiple regression, prediction intervals, and Gauss's method. The appendix includes explanations and examples as well as formulas. However, even after gaining a good understanding of these things, it is all too easy for a programmer to make a mistake when working out a range calculation by hand:

$$\text{Range} = t(\alpha/2, n-2)\sigma \sqrt{1 + \frac{1}{n} + \frac{(x_k - x_{avg})^2}{\sum_{i=1}^{n}(x_i - x_{avg})^2}}$$

or when writing a helper program to calculate correlation:

$$r(x,y) = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{\left[n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2\right]\left[n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2\right]}}$$

In the standard PSP curriculum, these particular equations are among those that students implement as they create PSP tools while completing the outlined assignments. The students then use the tools to do the PSP in future assignments, so if students don't accurately implement the equations, the very tools they use to do the PSP can introduce analysis errors.

It is also easy to make mistakes in choosing among the various statistical methods to use, particularly for size and time estimation. The user must also make decisions about which completed projects to include as historical data and which ones to treat as outliers.

Finally, it is important to remember that the PSP doesn't take place in a vacuum. The user has to deal with all the forms, scripts, instructions, and computations *at the same time* that he or she is carrying out another highly demanding intellectual task - software development. The part of the planning phase which contains the most difficult statistical steps occurs right between the conceptual design and detailed design steps. The Time Recording Log and Defect Recording Log must be filled out in the midst of the design, code, compile and test phases. This probably accounts for the high incidence of “stupid errors” in trivial calculations. Finding the number of minutes between start and stop times is not a difficult task, but for 4 out of 9 assignments, this mistake affected 7 to 9% of Time Recording Log entries. Programmers practice the PSP during development, and their attention is inevitably split between the two tasks.

7.3.2 Prevention

When considering ways to prevent analysis errors, the main answer is automation. Virtually every analysis error I discovered in the student data could have been prevented if the students had used a well-designed PSP tool. Such a tool could completely free users from the maze of phases, forms, scripts, and instructions. The tool could “understand” the process scripts well enough to guide the user through the phases, correctly ordering the forms and presenting only the needed fields in the proper order. The 275 blank field errors, 142 entry errors and 16 sequence errors point to the need for these features. The tool could, of course, do all calculations automatically. The 705 calculation errors show that this should be a primary requirement. The tool could provide context sensitive help at all times. It could handle all intra-project and inter-project data transfers and take over the management of project groups. A user would not have to remember anything about prior projects in order to do size and time estimation or view to date values. 212 inter-project transfer errors and 99 intra-project transfer errors show that this kind of functionality is necessary.

An all-in-one approach appears essential in designing a useful PSP application. During the PSP course, students had many tools available to them, ranging from calculators to spreadsheets to Java programs for size estimation and line counting. However, besides being a source of error, it is irritating, distracting, and time-consuming to use six or seven separate tools when it would be possible to seamlessly include every needed service within one package. An integrated approach could help to shift the user's focus from the complexities of "doing PSP", to actually looking at the data. When steps for the postmortem phase take two minutes, (as opposed to the average postmortem time in this study – about 90 minutes) it seems more likely that a user would take the time to really look at the results and think about the insights revealed. This kind of approach could also allow the user to see results of data analysis in many different formats and give the user access to raw data for further study.

Although Humphrey describes a manual approach to PSP in *A Discipline for Software Engineering* [9] and *Introduction to the Personal Software Process* [11], he also indicates that automation is desirable. For example,

> Tool support can make the methods described in this book more efficient and easier to use. Such standard aids as word processors, spreadsheets, and statistical packages provide an adequate base initially, but ultimately CASE environments are needed that embody the PSP methods in engineering workstations, in addition to all the other tools generally available (p. 26) [9].

Also, in reference to the collection stage,

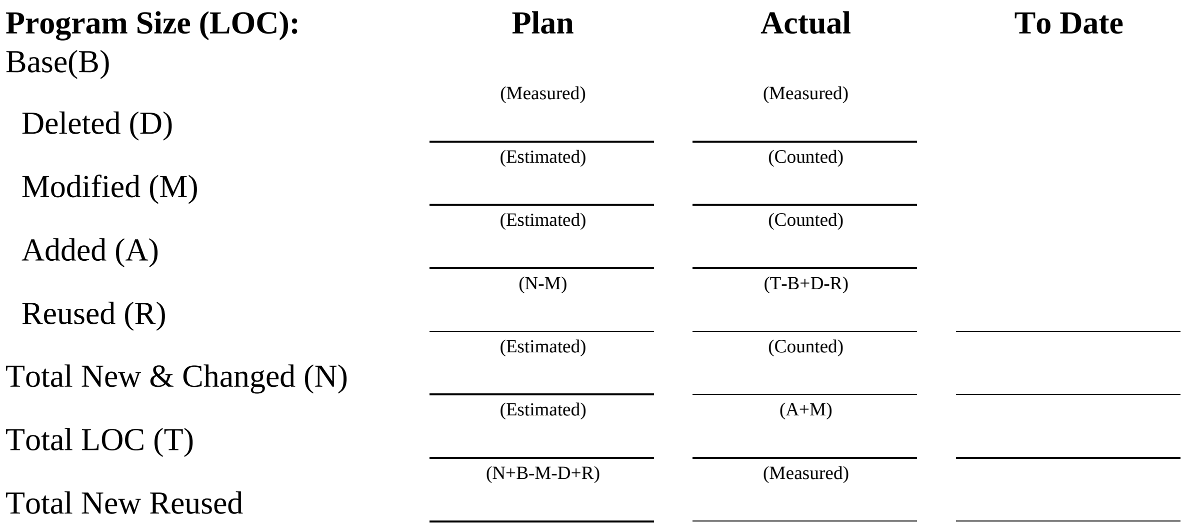
> Tools to assist in data gathering are feasible and could certainly help. They would probably not save a lot of time, but they could significantly improve data accuracy and completeness... Once the data are gathered and are in a database, many automatic analysis tools could assist in estimating, planning, and progress reporting (p. 219) [9].

Moving away from automation, what are other ways to prevent analysis errors? Manual PSP could be improved by making some simple modifications to the forms. Many fields contain numbers that must be transferred to other forms. Often the value in a field must be added to the value in the same field from a different project to provide a to date value. Students often correctly transferred a value from the wrong field when doing these kind of transfers. These errors could be prevented by visually linking the field being transferred from with the field being transferred to. For example, if *Program Size (LOC), Total New & Changed (N)* fields were printed as boxes drawn with a double line, and *Program Size (LOC), Total LOC (T)* fields were printed as boxes drawn with a dotted line, I believe it would cut down significantly on the transfer errors for size and time estimation. Figure 7.1 shows a portion of the current Project Plan Summary for PSP2. Figure 7.2 shows one possibility for formatting changes that could reduce the number of transfer errors.

Finally, to cut down on problems with finding and interpreting PSP instructions, it would be useful to have a PSP reference booklet for each process (such as PSP0, PSP0.1, etc.). The booklets could contain samples of all the forms used in the process and complete instructions for each one. They could also include process scripts and definitions for key concepts. Obviously a lot of the information would be duplicated from booklet to booklet, but only pertinent information would be included, and instructions could be much better ordered.

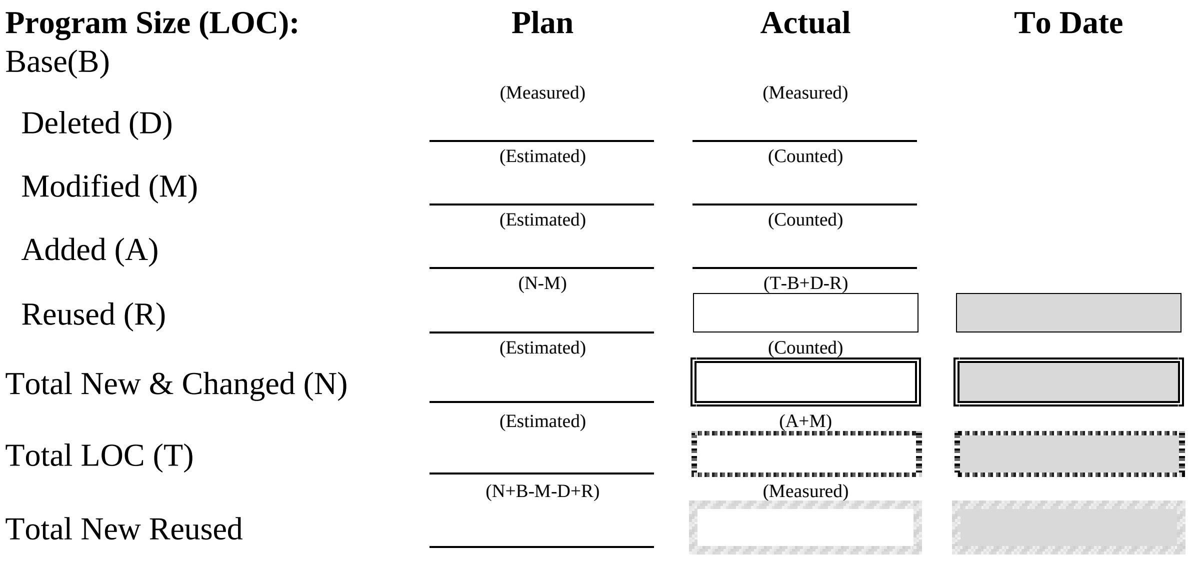
7.4 Evaluating PSP Data

Because of the questions this study raises about the quality of PSP data, I believe that PSP data should not be used to evaluate the PSP method itself. In other words, it is not 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 on PSP data alone, might only reflect a decreasing trend in the recording of defects, rather than a decreased trend in the defects present in the work product.



Program Size (LOC):	**Plan**	**Actual**	**To Date**
Base(B)	(Measured)	(Measured)	
Deleted (D)	(Estimated)	(Counted)	
Modified (M)	(Estimated)	(Counted)	
Added (A)	(N-M)	(T-B+D-R)	
Reused (R)	(Estimated)	(Counted)	
Total New & Changed (N)	(Estimated)	(A+M)	
Total LOC (T)	(N+B-M-D+R)	(Measured)	
Total New Reused			

Figure 7.1: *Portion of Current Project Plan Summary Form for PSP2*



Program Size (LOC):	**Plan**	**Actual**	**To Date**
Base(B)	(Measured)	(Measured)	
Deleted (D)	(Estimated)	(Counted)	
Modified (M)	(Estimated)	(Counted)	
Added (A)	(N-M)	(T-B+D-R)	
Reused (R)	(Estimated)	(Counted)	
Total New & Changed (N)	(Estimated)	(A+M)	
Total LOC (T)	(N+B-M-D+R)	(Measured)	
Total New Reused			

Figure 7.2: *Possible Modifications to Project Plan Summary Form for PSP2*

This is not to imply that all PSP evaluations are based upon PSP data alone. For example, in one of the case studies [6], 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 is not subject to PSP data quality problems.

It could be argued that this conclusion is invalid because Dr. Johnson did not focus on data quality during the course and that the study just revealed this isolated problem after the fact. After all, students will make errors, and are unlikely to improve without faculty guidance; with proper faculty guidance, error rates should drop significantly. However, as outlined in Section 4.1, Dr. Johnson addressed data quality even before the first lecture, and continued to focus on it throughout the semester. Also, before teaching the course, he had already taught the PSP for one semester at the graduate level and had instituted it within his research group, the Collaborative Software Development Laboratory. By the time of the study, he was quite experienced as both a teacher and a user of the PSP. Not only did he consider it to be one of the most powerful software engineering practices he had acquired in his career, but most of the students were very positive about the PSP in their post-course evaluations:

- "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."
- "[At the beginning of the course], 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."

- 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."

The projects I examined should have been of *at least* average data quality. Additionally, the analysis I did, even using automated tool support, was extremely tedious and time consuming, often requiring two or three hours for a single project. It is unlikely the average PSP instructor has the time or motivation to do this on a week-to-week basis.

It could also be argued that data quality problems are mainly confined to student projects. While it is true that students may be less motivated and less experienced (and therefore less accurate) than professional software developers, the most severe errors that are outlined in Section 6.2.1 are of the type that can happen to anyone. None of these error types are likely to be eliminated just by pointing out that the particular type of error is occurring. Repeatedly I found myself stumbling over these same errors (delta time calculation, inter-project and intra-project transfers) while trying myself to verify the student data. Moreover, it would seem that external pressures and distractions would be greater for software engineers in the field, and they could be even more likely to make these kinds of errors. Analysis of error age, covered in Section 6.1.3, showed that most PSP errors did not appear to be a function of learning how to do PSP. Instead, errors continued to occur in various measures many projects after the first introduction of the PSP steps that produced those measures. As far as collection stage errors, I can speak from personal experience and say that I still have days when I don't record all my defects or get messed up on time recording. There may be people who do perfect data collection, but for everyone else it is important to understand the areas in which problems can occur and the effect such problems can have on PSP measures.

7.5 Summary

In conclusion, this thesis describes a case study of PSP to assess data quality issues. It has been shown that PSP users can make substantial numbers of errors in both the collection and analysis stages, and that these errors can have a clear impact upon measures of quality and productivity. However, these results do not imply that the PSP method is wholly unuseful in improving time estimation and software quality. On the contrary, student evaluation of the PSP method was positive, and even if certain numerical values are incorrect, the process still provides students with profound new insight into the software development process. Instead, these results could be useful in motivating two essential types of improvement to the PSP process: attention to measurement dysfunction issues and integrated automated tool support. 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.

7.6 Future Directions

Although the PSP tool developed for this case study was adequate for the purpose of uncovering analysis errors, there is room for improvement, as outlined in Section 5.1.4. The tool could be expanded to include all forms and processes through PSP3 and could be enhanced to work more smoothly in some areas and to provide more flexibility when accessing outside applications such as text editors. When these improvements are complete, it would be possible to make the tool publicly available as open source software.

Since many people still do PSP manually or with limited tool support, it would be useful to make formatting changes to the PSP forms, as outlined in Section 7.3.2, and then empirically evaluate whether such changes help to reduce the number of analysis errors.

Finally, it would be very useful to attempt to isolate and study the effects of measurement dysfunction on PSP data. I do not know exactly how this could be done. This case study gives some idea of the scope of the analysis problem, but more work is needed to determine the severity and causes of the collection problem.

Bibliography

[1] Jody Armour and Watts S. Humphrey. Software product liability. Technical Report CMU/SEI-93-TR-13, Software Engineering Institute, Carnegie Mellon University, August 1993.

[2] Robert D. Austin. *Measuring and Managing Performance in Organizations*. Dorset House Publishing, 1996.

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

[4] Olof Dellien. The personal software process in industry. Master's thesis, Lund Institute of Technology (Sweden), Department of Communication Systems, November 1997.

[5] George A. Ferguson and Yoshio Takane. *Statistical Analysis In Psychology And Education*. McGraw-Hill Book Company, 6th edition, 1989.

[6] 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.

[7] Robert L. Glass. *Software Runaways: Lessons Learned from Massive Software Project Failures*. Prentice Hall, 1998.

[8] Will Hayes and James W. Over. The Personal Software Process (PSP): An empirical study of the impact of PSP on individual engineers. Technical Report CMU/SEI-97-TR-001, Software Eng. Inst., Pittsburgh, 1997.

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

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

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

[12] Watts S. Humphrey. Three dimensions of process improvement. part I: Process maturity. *CrossTalk*, February 1998.

[13] Watts S. Humphrey. Three dimensions of process improvement. part II: The personal process. *CrossTalk*, March 1998.

[14] Watts S. Humphrey. Three dimensions of process improvement. part III: The team process. *CrossTalk*, April 1998.

[15] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(1):129–130, January 1997.

[16] Steven P. Kness and Mark S. Satake. A Level 5 organization looks at the Personal Software Process. *CrossTalk*, October 1997. Description of introduction of PSP into a level 3 programming group. No results yet.

[17] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[18] Prof. J. L. Lions (Chairman of the Board). Ariane 5: Flight 501 failure. Report by the Inquiry Board. http://www.cnes.fr/actualites/news/rapport_501.html, July 19, 1996.

[19] Raymond R. Panko and Jr. Richard P. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Twenty-Ninth Hawaii International Conference on System Sciences*, 1996.

[20] PSP resources page at the University of Karlsruhe. http://wwwipd.ira.uka.de/%7Egramberg/PSP.

[21] Khalid Sherdil and Nazim H. Madhavji. Human-oriented improvement in the software process. In *Proceedings of the 5th European Workshop on Software Process Technology*, October 1996.

[22] B. Shostak. Adapting the personal software process to industry. *Software Process Newsletter No. 5*, Winter 1996.

[23] Andrew Worsley. What are the benefits of the PSP software process? http://www3.cm.deakin.edu.au/∼peter/PSP_data/talk.html, 1996.