Chapter 1

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, relate a few of my personal experiences with the PSP, and outline possible directions for future research. [1]

1.1 Further refinement of the PSP model

The next few sections will cover the causes and prevention of collection and analysis stage errors. As a summary of concepts covered in previous chapters, it will be useful at this point to expand the model shown in Figure **??** to outline the nature of the errors associated with each phase and their correction methods. [2]

- **Collection stage errors:** Occur in such fields such as Time Recording Log start/stop times, all Defect Recording Log fields, and program size measures.

[1] Don't panic about the footnotes - they are primarily included in an attempt to make the dull task of reading this chapter more interesting for you, and will all be removed before distribution outside of CSDL!

[2] Is this at all useful? (This represents some of my early thinking about error classifications.)

  - **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
    * 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, these can be reliably fixed by redoing 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.

1.2 The Collection Problem

In the two-stage model of the PSP illustrated in Figure **??**, the collection phase starts with actual work and ends with records of work. Problems in this phase occur when the records of work (size, time, and defect data,) do not accurately reflect the actual work done. As stated in Section **??**, 90 errors were found from the collection phase. 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 **??**, there are indications that the actual number of collection phase errors was much higher. Analysis phase errors are relatively easy to find, since the analysis steps can be duplicated and the resulting data sets compared. The collection phase 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 of 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) [**?**].

1.2.1 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 defects in the Defect Recording Log over time.

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 recorded. 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 phase, 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 phase 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 the line-counting program to the PSP form. [3]

1.2.2 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

[3] Stupid mistakes? At least programmers are better than users. An AST tech support person reported that one customer complained that her mouse was hard to control with the dust cover on it. The dust cover turned out to be the plastic bag in which the mouse was packaged... At Dell, a customer called to say he couldn't get his computer to fax anything. After 40 minutes, the tech discovered the man was trying to fax a piece of paper by holding it in front of the screen and hitting the "send" key... At Compaq, one customer was having diskette problems. After trouble shooting for a while (magnets, heat, etc.), tech asked the customer what else was being done with the diskette. Response: "I put a label on the diskette, roll it into the typewriter...".

by the computer. The application could also provide timing for defect fixes, 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 phase.

An automated version of the PSP could make doing PSP much simpler. This would free the user from some of the inherent distraction of using the PSP, leaving more mental room to remember to 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 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.

1.3 The Analysis Problem

In the two-stage model of the PSP illustrated in Figure **??**, the analysis phase starts with records of work and ends with analyzed work. Problems in this phase occur when PSP users make any kind of errors in this analysis, whether incorrectly performing computa-

tions, providing the wrong data for computations, or choosing the wrong statistical methods. As outlined in Section **??**, 1479 of the errors found in the case study were committed during the analysis phase. 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 **??**, steps taken to correct analysis phase 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".

1.3.1 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. [4] 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, on the PSP2 Project Plan Summary alone there are 177 fields.

Just having a lot of forms with a lot of fields shouldn't make things overwhelmingly

[4] PSP3? It would be easy to make errors just *counting* them: seven scripts, two checklists, and 16 forms.

A programmer once went to a school,
Thought to learn PSP - 'twould be cool.
But, oh!, all those forms,
Regression and norms –
She lit a fire with Humphrey as fuel.
(The book, not the man - don't worry.)

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. [5] 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 [**?**] 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

[5] And one should be careful about adding new forms or modifying processes: Q. How many workers at Rocky Flats, the former nuclear weapon components plant, should it take to change a light bulb?
A. Sixteen – and that's no joke. An internal memo written by a manager at the U.S. Department of Energy plant recommended a new safety procedure for "the replacement of a light bulb in a criticality beacon." The beacon, similar to the revolving red lamp atop a police car, warns workers of nuclear accidents. The memo said the job should take at least 16 people over 60 hours to replace the light. It added that the same job used to take 12 workers 4.15 hours.

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:

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

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. [6]

[6] A developer started to write

A routine for sorting - so trite.

But he used PSP

Couldn't focus, you see,

And it ended out taking all night.

1.3.2 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.[7]

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

[7] This tool should probably not be developed by the X Windows architects: "If the designers of X Windows built cars, there would be no fewer than 5 steering wheels hidden about the cockpit, none of which would follow the same principles - but you'd be able to shift gears with your car stereo, a useful feature at that."

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" [**?**] and "Introduction to the Personal Software Process", [**?**] 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) [**?**].

Also, in reference to the collection phase,

> 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) [**?**].

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*

fields were printed as boxes surrounded by a black line, and *Program Size (LOC), Total New and Changed* fields were printed as shaded gray boxes, I believe it would cut down significantly on the transfer errors for size and time estimation.

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.

1.4 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.

This is not to imply that all PSP evaluations are based upon PSP data alone. For example, in one of the case studies [**?**], 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 [8] 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 **??**, 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 [9] to be one of the most powerful software engineering practices he had acquired in his career, but most of the students indicated in post-course evaluation that they found the PSP to be very useful and interesting. 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 **??** 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 **??**, showed that most PSP errors did

[8] ... and has!

[9] ... at the time, anyway

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 phase 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.

1.5 Personal Experiences With PSP Automation

Automation of the PSP is part of the solution for both the collection phase and analysis phase problems. My own progress towards this goal was recorded in various journal entries over the eight months after my first introduction to the PSP. [10] Initially, I was not aware of the problems with PSP data; 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

[10] I think this section should be moved to the Tool chapter. It doesn't seem to fit in here very well, and this chapter is quite long anyway.

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 estimation] 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.

1.6 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 phases, 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.

1.7 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 **??**. 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 1.3.2, 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 measure-

ment 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.