Chapter 1

Overview

1.1 Software Disaster Stories

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 [5].

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[1].

In June of 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 [4]. 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 [6].

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 programmers have been programming, bugs and behind-schedule projects have been persistent problems.

Over the decades, the software industry has taken various approaches to addressing these challenges. At first, 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. 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, and these organization-level processes are, not surprisingly, more helpful to organizations than individuals. In the mid-90's the focus shifted back to the individual developer with the introduction of the Personal Software Process (PSP) as outlined in Watts Humphrey's 1995 book "A Discipline for Software Engineering" [3].

1.3 The Personal Software Process

The PSP is a self-improvement process for programmers. Each programmer makes himself or herself a longitudinal experiment of one subject. Each program written can benefit from data collected about past projects, and each program written can 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 higher-quality programs. By paying close attention to every defect made as well as overall patterns of errors, the PSP user becomes more aware of quality issues and is provided with structured mechanisms such as code review to help prevent similar defects from being made in future programs. Additionally, by tracking the points when each defect is injected and removed and the time required for removal, the developer is continually reminded that finding and removing defects early in the process is much less time-consuming and 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, a good estimate of the total time required for completion can often be made. This, of course, is very useful for planning schedules and budgets. Size and time estimation are done at the beginning of each project. As soon as some basic planning is done a similar set of past projects is selected and then various statistical techniques are used to compute likely size and time values for the current project.

On a simple level, when a programmer is using the PSP, he or she starts by doing planning work using a set of pre-defined worksheet-type forms. Then, while moving through the design, code, compile, and test phases, detailed records are kept on other forms 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 all the data collected about the project is analyzed and values such as *Lines of Code (LOC) per Hour* and *Defects per Thousand Lines of Code (KLOC)* are computed for both the current project and to-date (covering the set of similar projects used in planning combined with the current project). Therefore, at the most basic level, the PSP involves two main activities: collecting primary data such as size, defect, and time measures; and analyzing data to produce derived or to-date measures.

1.4 Motivation

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 [2]. Unfortunately, most of the data used to illustrate these positive trends about the PSP is actually data produced by the PSP itself.

By January 1997 I had been using the PSP for a full year for both academic and professional work. Dr. Philip Johnson (my thesis advisor) also used the PSP and had twice taught a full-semester class on the PSP. Although the general good results of using the PSP had been replicated in his classes, 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 mistakes people made while using the PSP and what sort of effect these mistakes would have on the measures produced.

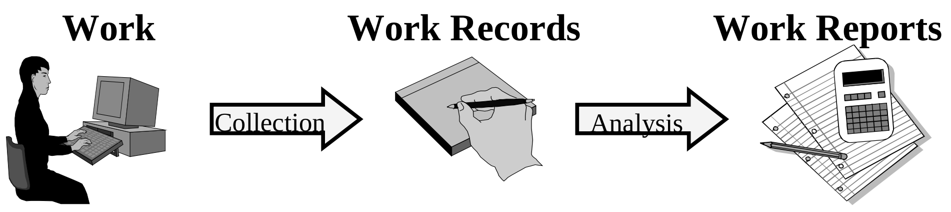

Figure 1.1: A simple model for PSP data quality. Through a process of *collection*, the developer generates an initial empirical representation ("Work Records") of his or her personal process ("Work"). Through additional *analyses*, the developer augments this initial empirical representation with derived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

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 "Work"—the actual developer efforts devoted to a software development project. As part of these efforts, the developer *collects* a set of primary measures on defects, time, and work product characteristics—the "Work Records". 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 "Work Reports"—and hopefully yield insights into ways to improve future software development activities.

This model was based upon the PSP as presented in "A Discipline for Software Engineering" 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, guides the user through the process steps, and aids in the collection of primary data.

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 insight into the programming process.

1.5 Hypothesis

We hypothesized that:

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

Resolving the collection problem requires addressing issues related to measurement dysfunction and the use of automated tool support. Resolving the analysis problem requires addressing issues related to the design and use of automated tool support. These two problems are severe enough to call into question the accuracy of current PSP results.

1.6 Case Study

Therefore, we decided to do a case study to evaluate PSP data quality, using student projects from one of Dr. Johnson's PSP classes. We wanted to determine what kinds of mistakes 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 mistakes is to be expected. Therefore, we felt it was important to evaluate not only the numbers and kinds of mistakes, 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.

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

To start with, I wrote a database application to automate a large part of the PSP and another application 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

I found 1539 errors in the class projects. When analyzing 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.

It could be argued 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 data field in which the error could be observed. 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.

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. 95.3% were found through the use of the PSP tool.

Comparisons of the original and corrected data showed 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

[I'll fill in this paragraph after finalizing the final chapter]

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 the PSP data from 90 software projects was evaluated, the methods used to correct errors in the PSP data, and how the results were recorded.

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.

Bibliography

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

[2] 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.

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

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

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

[6] 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.