Chapter 1

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!)

World-class programmers also want to improve their skills and could probably benefit from a similar learning cycle, but software development is very different from diving! We certainly 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. 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. Secondly, 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! In the software field, how is it possible to rate one software development effort over another? Do you use on-time completion, user satisfaction of the final product, efficiency of development, reusability, elegance of the design, quality of the documentation, or well-commented 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 pool 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 top-class software engineer.

So what does this have to do with the PSP? Well, for a motivated programmer the PSP can produce, in a virtual sort of way, a videotape of the development process. Following the structured forms and processes that the PSP provides, the programmer does actual work and simultaneously records 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. Secondly, 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.

1.1 PSP Levels

Now that the PSP has been covered on a conceptual level, the next few pages will describe in a more technical way its first five levels. These were the levels used in the PSP class which produced the projects reviewed for this research. First, however, it is important to understand that the PSP is really a set of increasingly complex processes called “levels”. When learning the PSP, a software engineer starts at PSP0 and gradually learns the more advanced processes one at a time. Secondly, 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 for PSP level 3 which is designed for large projects).

1.1.1 PSP0: Foundation

This baseline version of the PSP provides an introduction to the PSP environment of processes, forms, instructions, and process scripts. Six development phases are introduced: planning, design, code, compile, test, and postmortem. Three main forms are used: 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 produce 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 1.1 for a sample form.

1.1.2 PSP0.1: Measuring Size

The main difference between PSP0 and PSP0.1 is the addition of size measurement using LOC as the basic unit. The Process Improvement Proposal (PIP) form is introduced to record ideas about process improvement, lessons learned, and other notes. A coding standard is also added, 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

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 1.1: *Sample Project Plan Summary form for PSP0*

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, comments and ideas can be entered on the PIP form at any time. In the postmortem phase the user measures 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. To date totals are kept 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?"

1.1.3 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." [1] For object-oriented design, objects such as Java classes make good proxies. Once objects and methods have been determined in a conceptual design step, historical data about method size is used to determine a preliminary estimated size for the current project. Then historical data is examined, using techniques such as regression, to determine the relationship between planned size and actual size for prior projects. This is used to refine the size estimate for the current project. Historical data is referenced 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.

The planning phase is changed 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. The test phase is changed 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?"

1.1.4 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. New process statistics are also calculated on the Project Plan Summary.

The planning phase is enhanced 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?".

1.1.5 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: two new phases are introduced (design review following design and code review following code) and 71 fields are added to the Project Plan Summary form.

In planning the user now calculates the 70 percent 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. Sample design and code review checklists are provided, 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 testing?"

1.1.6 The Higher Levels

There are two more advanced process levels described in *A Discipline for Software Engineering* [1] 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.

1.2 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.

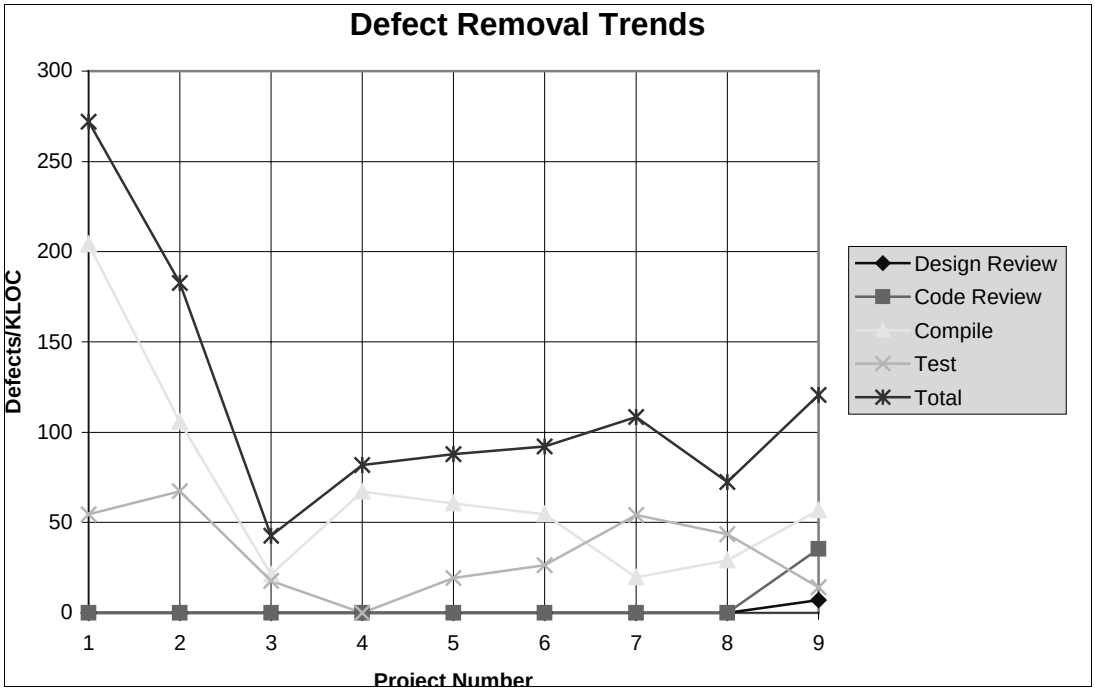

Figure 1.2: *Sample Defect Report*

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 1.2. 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.

1.3 User Modifications of the PSP

The PSP is described in a careful and detailed way in *A Discipline for Software Engineering* [1]. However, it is not Humphrey's intention to imply that this is the ultimate personal software process for any individual develo]per. 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.

Bibliography

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