THE UNIVERSITY OF CALGARY

Automated Support for a Custom Personal Software Development Process

by

Dale Couprie.

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE.

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 1998

0-612-34952-7

Abstract.

The Personal Software Process is a software development process that allows the individual to apply industrial level discipline to his or her practice. It is taught through a course at post-secondary institutions as well as selected companies.

While the PSP methods are helpful, many students are discouraged from using PSP. Its detailed record keeping carries a high overhead cost. In addition, software development tasks like requirements and documentation are not covered in PSP.

It is believed that the cost to perform PSP can be reduced through providing automated support for PSP activities. It is also believed that the PSP can become a basis for developing a custom personal software process.

The investigation focuses on the development and implementation of PSP Manager. It supports the PSP data collection activities and allows the user to adjust the default PSP to confirm to his software development practices.

Table of Contents.

List Of Tables.

List Of Figures.

List Of Abbreviations and Nomenclature.

COQ:	Cost of Quality.
CR:	Code Review.
CMM[1]:	Capability Maturity Model.
DR:	Design Review.
DRL:	Defect Removal Leverage.
IPD:	Institute for Program Structure and Data Organization.
KLOC:	One thousand lines of code.
LOC:	Lines of Code.
PROBE:	Proxy Based Estimation.
PIP:	Process Improvement Proposal.
PSP:	Personal Software Process.
SEI:	Software Engineering Institute.
TSP:	Team Software Process.

[1] CMM, PSP, TSP, "Capability Maturity Model", "Personal Software Process", and "Team Software Process" are service marks of Carnegie Mellon University.

Chapter 1. Aim and Objectives.

This thesis covers the motivation, design and implementation of an automated tool to provide automated support for a custom software development process. The tool will support the Personal Software Process (Humphrey, 1995a) while allowing the user to make modifications such that it fits his or her practice. The purpose of the tool is to reduce the overhead cost of performing the data collection and calculations required by the Personal Software Process, which can be executed with pen, paper forms and a calculator.

The availability of automated support would provide three benefits. First, automation would make data collection and calculation more efficient. Timing data could be collected through an automatic timer, allowing for automatic computation of time spent on the task. The automated tool can perform all calculations required by the Personal Software Process faster than doing each form by hand. Similarly, the automated tool can bring up any of the forms on request, as opposed to searching through papers to find a required form. Second, redundancy can be reduced. The tool can store each data item in one place and access it when it is required. This eliminates the need to copy data by hand between forms or from a form to a spreadsheet. Third, the tool can operate transparent to the project in the same multitasking environment as the project development environment. This transparency allows the user to switch between the tool and the development environment as opposed to going to the side to record data.

1.1. About the Personal Software Process.

The purpose of the Personal Software Process, developed by Watts Humphrey, is to introduce discipline to the individual software engineer. It does so through pointing out ones strengths and weaknesses through statistical measurements of one's work.

PSP consists of a series of scripts, forms, metrics, and methods that are used to collect software performance data. It is implemented in seven incremental levels. The next level consists of all methods in the current level plus one or two new features. Its structure is as follows:

- PSP 0: Establishment of a performance baseline.
- PSP 0.1: Software size measurement.
- PSP 1: Software size estimation, including the PROBE regression method.
- PSP 1.1: Resource, task and schedule planning.
- PSP 2: Defect prevention through code and design reviews.
- PSP 2.1: Quality assurance through design completeness.
- PSP 3: Cyclical development.

1.2. Aim.

The following statement is set as the aim of the thesis.

- **Aim**: To provide a software tool that implements automated support for the execution of a customized personal software development process using the Personal Software Process as a baseline in developing such a custom process.

PSP provides a structured set of tasks that are based on practices that have been successfully used in the software industry. In addition, it provides a metrics plan that involves the collection of time, defect, and software size data. The metrics plan requires detailed record keeping and calculation so problems in the process can be pinpointed. PSP is traditionally executed by hand using paper forms.

The derivation of this aim is based on problems that discourage programmers from using PSP. Given the intense record keeping of PSP, hand execution carries a high overhead cost. Through automated support, it is believed that the overhead cost of executing PSP can be reduced. In addition, hand execution of PSP forces the user to stop his programming task to record data. If automated support is provided, it can run on the same machine as his development environment, making data collection transparent to the task at hand.

Another problem with PSP is that it assumes the programmer is given a stable problem statement and is asked to develop and implement the software to solve the problem. In real life, this is uncommon. The software development life cycle covers risk management, formal specification, requirements, development, implementation, documentation and follow-up. During the software life cycle, changes in the problem statement can happen at any time. In addition, each software developer has different preferences, problems and practices. If automated support for PSP is provided, the user can make adjustments to the PSP such that his practices, strengths, and weaknesses are addressed.

1.3. Objectives.

To fulfill this aim, the following objectives must be fulfilled:

- **Objective 1.** To provide a tool that provides automated support for the Personal Software Process.
- **Objective 2.** To provide mechanisms that allow a user to adapt the tool to his or her personal software development process.
- **Objective 3.** To make the components of this software tool (except the spreadsheet component) portable between operating system environments.

If PSP is to be used as the baseline to develop one's customized personal process, the software tool must provide support for the PSP as documented by Humphrey. Once automated support is developed for the PSP, functionality can be added to the tool to make changes to the process based on personal needs.

In order for the tool to be effective, it needs to be conveniently available. The tool is most convenient if it runs in a multitasking environment. With the recent developments in the Internet and Java technology, the development of applications that can be ported between operating systems is easier. Since many operating systems are still in use, one can see why it is important to make the tool work on any operating system. This is possible if a portable application is developed.

1.4. Key Tasks.

However, the development of a tool will not of itself fulfill the aim and objectives. Four key tasks are defined.

- **Key Task 1**: The problems and challenges of the PSP must be identified such that the motivation to implement automated support for PSP is recognized.
- **Key Task 2**: Strengths and weaknesses of current PSP support tools must be identified.
- **Key Task 3**: The newly developed tool should overcome shortcomings of the existing PSP support tools.
- **Key Task 4**: The effectiveness of using the tool to execute the PSP process must be determined.

The need and demand for PSP automated support must be identified through an analysis of the literature available on the PSP. If there are any existing PSP automated support tools, they must be evaluated to identify strengths and weaknesses. The evaluations can also be used to identify use cases and requirements for the tool to be developed in this thesis. Finally, the completed software tool must be subject to a user study such that its effectiveness in executing PSP can be evaluated.

1.5. Summary.

This chapter provided the aims, objectives and key tasks of this thesis. Given the existence of a disciplined individual-level software process in the PSP, it makes sense to derive a customized personal process based on the principles of the PSP. Thus, the thesis and its accompanying tool will use the PSP as a baseline.

1.6. Thesis Structure.

Chapter 2 provides a detailed description of the Personal Software Process framework, standards, costs and benefits. The chapter consists of how the PSP fulfills the key practice areas of the Capability Maturity Model. The chapter also discusses its costs, benefits, and impact. Four tools that provide automated support for PSP are also discussed.

Chapter 3 provides a critical review of the Personal Software Process, including its advantages and disadvantages. It includes some critical remarks on the research done on the impact of PSP as well as reviews of the four automated PSP support tools introduced in Chapter 2.

Chapter 4 contains the results of the requirements for a customizable PSP support tool. The requirements are elicited using a use case analysis.

Chapter 5 provides an overview of the design of the tool. It includes a layout for the PSP user's profile as well as a design for the spreadsheets and reports that will be produced by the tool. It wraps up with the design of the user interface and menu layout.

Chapter 6 provides an implementation level view of the tool's operation. Of particular interest are the lists of collected data, their manipulation, and the queries that are performed on each list. Chapter 6 also includes how time spans are stored, how a project is created, and the persistency of the user's profile.

Chapter 7 provides an evaluation and critical review of the completed tool. The evaluation includes a tracing back to the requirements, key tasks, objectives and aim.

Chapter 8 provides a conclusion to the work and provides directions for future research for both the PSP and automated support development.

The thesis contains two appendices. Appendix A consists of the requirements for a programmer profile to allow the tool to support the actual PSP while appendix B shows the layouts of the PSP project and career spreadsheet reports.

Chapter 2. A Review of the Personal Software Process.

Watts Humphrey of the Software Engineering Institute developed the Personal Software Process (PSP). It is geared specifically to the individual software engineer, allowing him to apply an industrial level discipline to his software development skills.

PSP is introduced with a course consisting of fifteen lectures, ten programming assignments, and five reports (Humphrey, 1995a). The course is effectively taught at the graduate level. It is also effectively taught in industry provided the management shows their support (Humphrey, 1995b).

While PSP can be enacted as a software process, it is best when it serves as a framework for building discipline into one's software development practice. The engineer starts with this framework and makes adjustments to suit his own personal needs, practices and common mistakes.

This chapter will document the incremental structure of the Personal Software Process. It will also include a description of each of the PSP forms and their applicability in the software development process. The costs, benefits, and impact of the PSP will also be documented. The relationship between the PSP and the Software Engineering Institute's Capability Maturity Model (CMM) is also investigated in this chapter. It was found that not all of the key practice areas of the CMM could be applied at the personal level.

The chapter wraps up with an investigation of four software tools that can be used to provide automated support for the execution of PSP. Each of the tools has its benefits and limitations. These tools will serve as a foundation for the development of a software tool to support a custom personal software process.

2.1. The Objectives of the PSP.

Evidence has been recorded that the PSP is an effective process for software development (Hayes and Over, 1997). It was derived from principles proven in the software industry (Humphrey, 1995a). The following are objectives of the PSP.

- The definition, measurement and tracking of work will help software engineers become more understanding of what they do.
- The PSP provides a defined process structure and measurable criteria, providing feedback to the software engineer.
- The feedback is explicit, supporting continuous process improvement.
- The defined process structure of the PSP can improve working efficiency.

2.2. The Structure of the PSP.

2.2.1. Motivations.

Watts Humphrey (Humphrey, 1996a) identified a gap between an individual's small projects and the corporation's large-scale projects when it comes to the Capability Maturity Model (CMM). They could not see how the CMM could apply to small projects.

CMM applies to all sizes of projects, however guidance for small projects needed to be more detailed. This brought about the need to bridge this gap between organizations and individuals. Humphrey conducted an investigation into how individual engineers and small organizations could apply the CMM Key Process Areas was conducted. The investigation resulted in the methods used in PSP (Humphrey, 1996a). They cover 12 of the 18 key process areas. The initial reaction to the PSP was positive. Experienced engineers realized that the PSP methods helped them in their work environment (Humphrey, 1996b).

The PSP also addresses a gap between university training and the software engineering industry. Post secondary education in computer science covers concrete "product" topics like programming languages, digital logic, algorithms, and graphics. These topics would be considered the "meat and potatoes" of software engineering. It is also noted that university has a focus on individual contribution and that many academic projects are small by industry standards (Hilburn and Towhidnejad, 1997). The industry expects new software engineering graduates to know about teamwork, process, quality and large-scale projects. These skills include the negotiation of requirements, planning, estimation of cost, and quality assurance. Of these skills, only the negotiation of requirements is not covered in the PSP (Hilburn and Towhidnejad, 1997).

2.2.2. Structure.

The PSP can be performed in any one of seven different levels (Humphrey, 1995a). A baseline measurement is done at level PSP0 with Each subsequent level adding software development activities, forms and metrics.

2.2.2.1. The Phases of PSP.

Shown in Table 1 are the basic phases in the PSP. A phase is defined as a task element in a software process. The PSP proceedings fit the waterfall model for software development, where tasks are done in a specific order with no repeating. In the real world, the waterfall model is not effective (Humphrey, 1989). Through the cyclical development process in PSP level 3 (Humphrey, 1995a), the more effective incremental life cycle can be implemented.

2.2.2.2. PSP Plan Summary.

The cover form for the PSP is the Project Plan Summary, which summarizes the data collected during the project. The nature of the data collection allows time and defect statistics to be organized by phase as in Table 2. The plan summary also includes data on software size as well as additional metrics computed from the time and size data. Since there are changes in the metrics plan at each level of PSP, there is a separate Plan Summary form for each level.

Phase Name	Purpose
Planning	A strategy for developing the program is made, including time, cost, and resource estimates. A schedule for the project is also created.
Design	A "blueprint" of the software is drafted in a specified format (e. g. UML diagrams).
Design Review	The design document is reviewed against a checklist of design guidelines. Any defects found are recorded and fixed.
Code	The software is implemented in a development environment.
Code Review	The code is inspected against a checklist of coding guidelines. Any defects found are recorded and fixed.
Compile	The code is compiled, fixing and recording all defects.
Test	Tests are performed on the compiled code, fixing and recording all defects.
Post-Mortem	When all the test cases have succeeded, this phase is used for reporting and summarizing process data collected during the project.

Table 1. The basic phases of PSP

2.2.2.3. PSP0: The Baseline Personal Process.

The PSP0 process allows users to gather data on their own work such that they gain a quantitative understanding of their work. Since the use of PSP0 will set a benchmark for improvement, it is important to follow the process closely. PSP forces the programmer to restrain himself and learn based on what data is retrieved from the process (Humphrey, 1995).

Phase	Estimated Time in Phase (min)	Actual Time In Phase (min)	Time To Date	To Date (%)
Planning	13	18	119	11.28
Design	35	41	237	22.46
Code	33	37	218	20.66
Compile	15	14	99	9.38
Test	28	38	199	18.86
Post Mortem	30	32	183	17.35
Total	154	166	1055	100.0

Table 2. Sample Plan Summary Table for Time.

The Plan Summary Form for PSP0 contains three tables for time in Phase, Defects Injected by Phase, and Defects Removed by Phase. Filling in these three components is done through the aid of two key PSP forms.

Date	Start	Stop	Interruption Time (min)	Time Elapsed (min)	Phase	Comments
July 8, 98	3:14	3:53	0	39	Planning	
July 8, 98	3:53	4:48	16	39	Design	Fire alarm.

Table 3. Sample Time Recording Log

The *Time Recording Log* (shown in Table 3) provides a strict guideline for the recording of time spent in a project. It is essential to track interruption time as time spent in disruption is time lost and should not be counted as time on the task. Time may be tracked using any timepiece. The Time Recording Log uses minutes for precision.

The *Defect Recording Log* (shown in Table 4) provides a standard for collecting defect data that is equally as strict as time recording. A defect is defined as an error that was introduced (injected) in one phase and removed in a later phase. An error such as typographical error that is immediately corrected as it happens is not a defect. Each defect found constitutes one entry in the defect recording log.

Date	Defect No.	Defect type	Phase injected	Phase removed	Fix Time	Description
July 12, 98	1	20	Code	Compile	2	Missing brackets.
July 13, 98	2	50	Code	Test	14	Incorrect parameter.
July 13, 98	3	50	Code	Test	6	Found while fixing defect 2- incorrect parameter.

Table 4. Sample Defect Recording Log.

Type Number	Defect Type Name
10	Documentation
20	Syntax
30	Build, Package (change management)
40	Assignment (limits, declarations)
50	Interface (procedure calls, parameters, I/O)
60	Checking
70	Data
80	Function (loops, logic, recursion)
90	System (configuration, memory)
100	Environment.

Table 5. Defect Type Standard.

If a second defect is found while fixing the first defect, it can be acknowledged as shown in the last entry of Table 4. The definition of the phase in which a defect was injected is subjective. However it is up to the user to determine the phase in which the defect originated. The phase of removal for a given defect is the phase in which it is found and fixed. When code changes are required to fix a defect in the test phase, the time spent editing and compiling the changed code is counted as testing time. The type of a defect is selected from the Defect Type Standard (shown as Table 5). Defect types can be further sub typed to match a user's most problematic areas.

2.2.2.4. PSP0.1: Software Size.

PSP0.1 introduces the concept of software size. Logical Lines of Code is used as the unit of measurement. This allows the user to divide the code into various size categories. The PSP provides a coding standard that allocates one logical line of code per physical line in the file.

In the Plan Summary, the total time estimate is divided among the phases based on the Programmer's historical time spent in the phases. The planning phase also consists of an educated guess of the software size. The following software size categories are introduced:

- *Base*: If the program is to be written from scratch, this metric is zero. If an existing program is being modified, it is set to the measured size of the existing program.
- *Deleted*: Lines of code that are removed from the base program are tallied.
- *Modified*: Lines of code that are changed in the base program are tallied.
- *Reused*: The PSP defines reused code as code written by the programmer in a previous project that is used in the current project. Code that is part of a standard library such as input and output calls are not counted. Each reused module is measured separately.
- *Added*: New lines of code are counted as added code. To compute the amount of new code, measure the total size of the program, subtracting the base and reused lines of code, and add the number of deleted lines (to account for the lines of code that were replaced).

- *New and changed*: This is the sum of the modified and added code. It is the official software size metric for PSP because development time is focused on developing these lines of code (Humphrey, 1995a). Base and reused code is already developed and ready for use.
- *Total*: The total size of the final program is measured using a software measurement tool.
- *New Reusable*: Any newly created module that may be reused in a future project is measured.

The *Process Improvement Proposal* (PIP) form is introduced in PSP0.1 to allow the user to track problems with his personal process. The form documents a problem and its proposed solution. Each problem is assigned a unique ID. Programmers are not required to find an immediate solution to a process problem, but recording problems on a form increases programmer awareness of the problem.

2.2.2.5. PSP1. Size Estimation.

PSP1 formalizes the estimation of the size of the program. It introduces PROBE (Humphrey, 1995a), which is an algorithmic based method for the estimation of size. PROBE analyzes the historical database and looks for a trend in the relationship between estimated and actual size. It assumes that past performance is indicative of future performance. If the historical database shows that the final project size is traditionally higher than the programmer's original estimate by some percentage, PROBE (graphically shown in Figure 1) allows the user to enlarge his original estimate by that percentage. It uses a linear regression analysis (Leabo, 1968) on the relationship between the estimated size and the actual size. The equation of the best-fit line is used to convert an initial estimate to a projected estimate. Through the use of the Student's T distribution (Leabo, 1968), a bound on the error of estimation can be placed for a given confidence level. If the linear relationship between data points is weak, PROBE will provide a result with a high range of error.

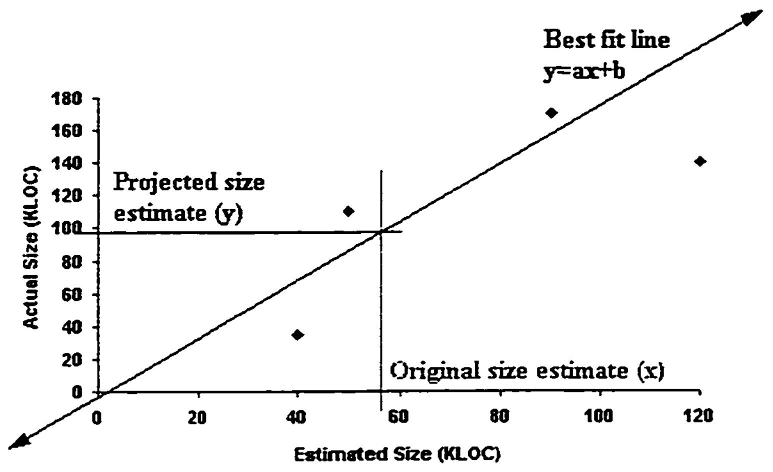


Figure 1. Graphical representation of PROBE.

PROBE stands for Proxy Based Estimation. A proxy is a substitute measurement unit that can be related to the actual measurement unit used in measuring the size of the final product (Humphrey, 1995a). Since lines of code is difficult to visualize early on in the project, programmers may opt to estimate size in function points or the number of objects.

PSP1 also introduces the *Test Report Template*. Each template represents one test case that consists of identifier number, objective, inputs, expected outputs, and actual outputs. A productivity metric is also introduced, measured as *Lines of Code Per Hour* (Humphrey 1995a).

2.2.2.6. PSP1.1. Tasks and Scheduling.

PSP1.1 provides scheduling and tracking facilities. A ratio called *Cost Performance Index* (Humphrey, 1995a) is added, which is the ratio of planned time to actual time. The programmer's goal is to have a CPI slightly above 1. A CPI below 1 indicates the programmer has a delay problem while a CPI well above 1 implies plans are too pessimistic.

Two key forms are introduced. The *Task Planning Template* allows a project to be divided into tasks (usually PSP phases). Each task has a description and is allocated some time. Two key metrics called *planned value* and *earned value* are introduced in the task template. Each task is assigned a planned time. A planned value for a task is computed as the percentage of the planned time for the project. Planned Value accumulates as tasks are added. As tasks are completed, the planned value for that task is added onto the current earned value for the project. No earned value is accumulated until the task is complete. The companion to the task template is the *Schedule Planning Template*, which is organized by the day. The programmer uses this template to plan tasks and estimates the earned value of project tasks to be completed every day. Tracking the project is done daily by comparing the expected earned value to the actual earned value.

2.2.2.7. PSP2. Defect Prevention.

The later in the process a defect is found, the higher the cost for its repair (Humphrey, 1997). For example the compiler could miss a simple coding mistake that propagates to a challenging defect in the testing phase. That same defect would take a minute to fix in a review. The concept of reviews encourages programmers to find and fix defects before going on to the next phase of development. A small amount of time budgeted to reviews can lead to significant savings at testing time (Humphrey, 1997).

Design and code reviews are introduced as new PSP phases because they are significant tasks apart from development. A checklist accompanies each review. The PSP provides standard checklists for the code and design reviews. However, programmers are encouraged to extend and change checklists such that they match one's strengths and weaknesses.

PSP2 introduces two metrics. *Yield* is defined as the percentage of the total product defects that were removed before or during the code review phase. Its purpose is to measure the effectiveness of code and design reviews. The second metric is called

Defect Removal Leverage and it measures the ratio of defect removal rates for two phases. Its purpose is to show the programmer the higher efficiency of finding defects in review phases than testing.

2.2.2.8. PSP2.1. Design Completeness.

PSP 2.1 requires the developer address design completeness and consistency. It introduces four design templates: operational scenarios, state specifications, logic specifications, and functional specifications. These templates serve as guidelines for the different components of a software design document and prove useful in a design review.

The *Cost of Quality* (COQ) metric introduced in PSP2.1 is defined as the percentage of the development time spent evaluating the software product for defects (Humphrey 1995a). It is divided into two levels. The *Appraisal COQ* covers the time spent appraising the product through design and code reviews. The *Failure COQ* covers the time spent during the compile and testing phase. The Failure COQ metric is named as such because defects found in compile and testing are found as a result of compilation and program failures respectively.

2.2.2.9. PSP3. Cyclical Development.

Most large-scale projects cannot be practically done using the waterfall or V shaped life cycle that the PSP is based on. While the PSP can be fitted as part of the spiral life cycle (Scacchi, 1987), PSP must be adjusted to support the incremental life cycle. PSP3 is designed to address the incremental life cycle by subdividing the project into components that can be performed using PSP2.1. The opening cycle consists of developing a base program, and each subsequent cycle produces a major enhancement to the product from the previous cycle.

PSP3 divides design into high and detailed levels, with review phases at both levels. The high level design is normally performed once, while detailed design happens every cycle.

Project design issues will be disregarded unless they are documented. For this reason, PSP3 introduces the *Issue Tracking Log*. These design issues should not be confused with process problems. Each issue has an identifier, a date found, a phase in which it was found, a description, a date of resolution, and a description of its resolution.

2.2.3. Coverage of the Capability Maturity Model Key Process Areas.

The Capability Maturity Model (CMM) (Paulk, Curtis, Chrissis, and Weber, 1993) provides a framework of five maturity levels and 18 key process areas for continuous improvement in the software process. Improvements in these steps do not require extraordinary breakthroughs. A series of small adjustments and changes tend to be effective in improving a company's software process.

Humphrey used the CMM as a foundation for defining the PSP (Humphrey 1995b). He started through identifying 12 of the 18 CMM key process areas that could be applied at an individual level. Table 6 shows how the PSP supports the CMM Key Practice Areas. However not all CMM Key Process areas can be applied to the PSP. For the following six Key Process Areas, it is difficult to prove they would be effective using small programs (Humphrey, 1996a).

- Requirements Management.
- Subcontract Management.
- Quality Assurance.
- Training.
- Inter group Coordination.

The definition of a small program is subjective as the threshold for engineers may vary from 1,000 to 10,000 lines of code.

Humphrey says configuration management cannot be demonstrated at an individual level (Humphrey, 1995a). However, it is up to the individual programmer to implement his own strategy for configuration and version management.

Level	Key Process Area.	Applicability in PSP.
2. Repeatable.	Software Project Planning.	Starting at PSP0.1.
2. Repeatable.	Software Project Tracking.	Task Planning and Schedule Planning Templates.
3. Defined.	Peer Reviews.	Individual code and design reviews (PSP2).
3. Defined.	Software Product Engineering.	Test Reports (PSP1).
3. Defined.	Integrated Software Management.	Issue tracking (PSP3).
3. Defined.	Software Process Definition	All PSP steps are defined and documented.
3. Defined.	Software Process Focus	PIP (PSP0.1).
4. Managed.	Quality Management.	Reviews (PSP2), design completeness (PSP2.1).
4. Managed.	Quantitative Process Management.	Software metrics starting in PSP0.1.
5. Optimizing	Defect Prevention.	Reviews (PSP2), data analysis, checklist enhancements.
5. Optimizing	Technology Change Management.	Automated support, test reports (PSP1).
5. Optimizing	Process Change Management.	PIP (PSP0.1)

Table 6. PSP Coverage of the Key Process Areas of CMM.

2.3. The Cost of PSP.

Applying the PSP to a practice has a significant cost and a high degree of commitment (Humphrey, 1995a). First, it takes *time* to learn and apply its methods. Learning PSP can take up to four months. The course work for PSP will take an average of 130 man-hours (Humphrey, 1995a). After you have learned the process methods, it still will take time to collect and analyze data. As process improvements are implemented, they will take time to get used to. The time spent applying PSP will vary depending on the tools you are using to apply the methods. In the case of pen and paper, it can take a minute to log each time entry and each defect. When the project is finished, it can take up to an hour to gather all the data together and compute your performance metrics (Humphrey, 1995a).

Secondly, PSP can be challenging on your *emotion*. Many expect an instant improvement from PSP, but it does not work that way. While people who are new to software process generally see increased productivity, the engineer with a stable method who learns PSP will experience a decrease in productivity (Hayes and Over, 1997).

Finally, PSP brings a risk of conflicting with one's *self image*. One must recognize his or her strengths and should never dwell on his or her weaknesses.

2.4. The Benefits of PSP.

While the PSP has a number of costs, a large number of benefits can be realized when one's personal process has been stabilized (Humphrey, 1995a). They are listed as follows:

- The programmer gains an appreciation of his strengths and weaknesses that show in PSP data.
- The programmer may interpolate collected data such that ideas for process improvement can be derived.
- A programmer can resist an unreasonable pressure through discussion of the anticipated size of the problem and relating it to his or her historical productivity.
- The organized completion of a project gives a programmer a sense of personal accomplishment.
- Since the programmer has a repeatable, consistent and stable process he will received an increased confidence from his team membership.

2.5. The Current Impact of PSP.

The PSP is a relatively new technology, however it is making an impact. Those taking the PSP course are noting improvements. In addition, its methods are being applied to both industry and undergraduate studies.

2.5.1. Improvements noted in the PSP Course.

As engineers proceed through the PSP course (Humphrey, 1995a), they produce analyses on their progress. Many students appreciated the improvements the course brought to their software development process. Two experiments were performed.

Hayes and Over analyzed improvements of students as they made progress in the PSP course. The findings were filed in a technical report with the Software Engineering Institute (Hayes and Over, 1997). They performed an analysis of defect density, estimation accuracy, defect yields and productivity across the three PSP phases. The data used in the report was collected from 298 students in 23 groups.

Hayes and Over tested the null hypothesis that there was no improvement in a specific metric against the alternative that there was an improvement. The statistical confidences in the Hayes and Over report were communicated through the probability of a Type I error. A Type I error (Leabo, 1968) occurs when it was concluded an improvement was made in a data value when in fact there was no improvement in that data value. The following analyses were performed:

- A statistically significant reduction in overall defect density was noticed as the students proceeded through the course. This reduction was most noticeable as the students made the transition from PSP0 to PSP1 ($p < 0.0005$).
- The students recorded improvements in their size estimates. The improvement was most notable at PSP level 1 when the PROBE method for size estimation was introduced ($p = 0.041$).

- The students recorded improvements in their time estimates. The improvement was most notable at PSP level ($p < 0.0005$).
- Students noticed an improvement in the location of defects (yield) before the first compile. However, students did not show any improvement in this measurement until the introduction of code and design reviews at PSP2 ($p < 0.0005$).
- Hayes and Over could not conclude that student productivity increased through the PSP course. However, they also could not conclude that executing PSP caused a loss in productivity. Productivity showed minor fluctuations between the PSP0, PSP1, and PSP2 levels.

Humphrey also wrote a report on improvements noted in the PSP course (Humphrey, 1995b). His analysis covered 104 students in eight groups. Humphrey noticed the following impacts.

- Humphrey noted a significant improvement in defect density ($p < 0.005$), backing the findings of Hayes and Over.
- Defect yield improved significantly as engineers proceeded through the course time. As with Hayes and Over, no significant improvement was made until code and design reviews were introduced in PSP2.
- Humphrey found that inexperienced programmers had an improvement in their productivity as their defect rates were reduced. Similarly, experienced programmers showed a decline in productivity, as they had to add the extra overhead of doing tasks required by PSP. Humphrey concluded that maximizing productivity is not of itself a benefit to an engineer. While the PSP tasks take time, proper performance of the PSP tasks lead to higher quality products and lower test time.

2.5.2. Applying PSP Methods to Undergraduate Courses.

Universities are introducing PSP methods in their undergraduate academic program (Hilburn and Towhidnejad, 1997; Williams, 1997; Bagert, 1997). The university believed that using the methods of PSP would allow students to develop the foundation for disciplined software development, which is being sought by the industry (Hilburn and Towhidnejad, 1997). Hilburn believes that the PSP practices need to be introduced into the program at the beginning of the program, otherwise students would develop poor programming habits.

Concepts like time management can be introduced at the first year and continue throughout the program (Hilburn and Towhidnejad, 1997). Defect recording and prevention was also covered (Williams, 1997). The methods brought a mixed reaction from students. Students recognized that the concepts would benefit them at both the academic and professional level. However, they did not like continuous recording of common defects such as typing errors. (Williams, 1997).

A common interest among the three authors was the demand for automated support (Williams, 1997; Hilburn and Towhidnejad, 1997; Bagert, 1997). The PSP involves a high level of record keeping and involves a lot of data as well as calculations (SEI, 1996). This additional overhead is not well accepted by students and can provide students with a bad impression about both the PSP and software engineering (Hilburn and Towhidnejad, 1997).

2.5.3. Industrial Application of PSP.

Three companies have participated in the Software Engineering Institute's PSP industrial introduction strategy. Currently, experience data from the companies are not available. However, engineers have found that the PSP approach is useful for their work (Humphrey, 1995b).

If the PSP is to be successfully introduced into an industrial environment, it must involve a full commitment from the entire company (Humphrey, 1995a). It should include a dedicated instructor and resources. The involvement of the management is paramount. They need to motivate the engineers to do the PSP work (Humphrey, 1995a).

If one is a solo user of PSP in a corporation, he will be challenged to remain consistent. He should not discuss PSP with colleagues until he is comfortable in his process. He may be subject to persecution from peers regardless of his experience record. Other colleagues may feel threatened if his performance is better than they are. If little change is noted, he may be subject to ridicule. (Humphrey, 1995a).

2.6. Why develop automated support for the PSP?

The author notices four important drawbacks to the pen and paper approach to maintain PSP data.

- Using pen and paper to execute PSP is tedious.
- The human is naturally error prone.
- Paper forms are "straight jacketed" as they do not allow the user much room to deviate from the prescribed process.
- A correction in one form will force the recalculation of many fields, which takes a lot of time by hand.

These drawbacks can be alleviated if the process could be automated. Through the use of a PSP support tool the following benefits can be realized:

- The collection of timing and defect data would be simplified.
- The calculations required by the PSP are computed quickly.
- Data forms can be accessed on demand.
- It would be unnecessary to duplicate data, making data consistent between forms.
- Historical data would be stored conveniently for analysis.
- An automated guidance on the task sequence could be easily followed.
- Mistakes can be easily corrected through editing facilities.

- The precision of project data is limited to the computing power of the platform.
- A user could easily customize the PSP to his own practice.

Automated support would improve efficiency during development if timing is automated. Efficiency would be noticeably improved in the finishing stage of a project where performance data is calculated and posted. However, a poorly developed tool would be an inconvenience to its users (Humphrey 1995b). A PSP support tool should be transparent with respect to the task at hand such that engineers can focus on their products instead of their process (Williams, 1997).

2.7. PSP Automated Support.

The remainder of this review is a summary of the PSP tools currently available.

2.7.1. psptool 0.6.

psptool 0.6 (Wolesley, 1997) is a platform independent TCL-TK script written by Andrew Wolesley. It provides an implementation of PSP 2.1. The tool features a main control panel, which manages the project plan summary, creates bug instances, and tracks time.

The tool implements timing using a start-stop button. Time is measured in seconds. The accumulated time in the current phase is shown and can be edited. A time log entry is made and the time resets to zero whenever the phase is changed. The state of the timer does not change when the phase is changed. If it is running, it will stay running. The user may work in any phase at any time.

The tool supports the standard defect types and allows the users to create their own defect types. Many defects can be outstanding at the same time. When the timer is running, the time collected by the timer will also accumulate in each outstanding defect as that defect's repair time.

The Plan Summary window can be summoned at any time. Size estimates can be edited at the planning phase and the official software size can be edited at the post-mortem phase. Otherwise, the plan summary is read only. The window automatically updates as time and defect entries are added.

2.7.2. Timelog.

Timelog (Clemens, 1997) is a Java based implementation of the PSP Time Recording Log, written by Christoph Clemens as part of a Masters thesis. It implements all the standard components of the PSP Time Recording Log. In addition, it calculated the "Time In Phase" metric required by PSP. However, it does not automatically create time recording log entries as the timer is started and stopped. Both the Start and Stop time fields feature a "now" button which sets that field to the current time. It is up to the user to determine that the times and phases are properly set before explicitly adding the entry to the time log. There is also an interruption field that is toggled through a pause button.

The proper procedure for creating a time entry in this tool is as follows:

- Press "Now" by the Start field to get the current starting time.
- Execute your task.
- If disrupted, press the Pause button by the interrupt field. A pop up window tracks the interruption time. Press OK when the disruption is over.
- When finished the task, press "Now" by the Stop field to get the stopping time.
- Select a phase and write in your comments.
- Press "Add" to add the time log to the entry.

There is always one entry in the list that is highlighted. There are two important reasons for this. All entries added are added after this highlighted entry. This feature also gives the user the power to carry on from his last entry. A "continue" button associated with the start time fills the field with the finishing time of the highlighted entry.

Timelog supports the addition, deletion and renaming of phase names. In addition, the user may maintain more than one list of phases.

2.7.3. Timmie.

Timmie (Institute for Program Structures and Data Organization, 1997) is a time tracking tool available through the IPD Group at the University of Karlsruhe. It is a program that tracks time spent on different projects. It is a Java derivation from the UNIX application Titrax (Institute for Program Structures and Data Organization, 1997). Timmie supports sub-projects by separating the main project name from its sub-project name with a period. While this is not specifically a PSP tool, it can be manipulated to support the PSP by naming one project in the name of each PSP phase. Timmie can also track events in each project.

Timing is automatic and many projects can be managed. When the window is active, all current projects are shown and time automatically accumulates for the project that is highlighted.

Timmie comes with a Perl script called *accumtimmie*, which sums up time spent on certain projects from its log files. It requires a command line call with project names (PSP phase names in our case) and a time span to produce a report on the total time spent during the time span on the projects specified.

2.7.4. PSP Studio.

PSP Studio (Henry, 1997) was developed in 1997 as a project of the Department of Computer Science at East Tennessee State University. It runs on the Microsoft Windows platform and is driven using the database management system SQL Anywhere. While there are some cosmetic bugs with PSP Studio, this product supports more PSP tasks than the other PSP support tools. It is specifically designed for use with the PSP course.

The most noticeable attribute of PSP Studio is how it can be adapted to perform the PSP at any of the levels. It can support PSP at level 0 as well as at level 3. When

PSP Studio supports PSP below level 3, the tasks, features, and forms implemented at the higher levels never interfere.

PSP Studio uses tab-set navigation to categorize the forms. *Log forms* cover the time, defect and issue data collection. Each of the logs can be edited. *Templates* cover task planning, schedule planning, design templates, and size estimation. *Summaries* include the Plan Summary for a specific level of the PSP. In PSP level 3, a Cycle Summary form is also implemented. *Checklists* are available for the design review and code review phase. They start out as blank. PSP Studio implements only one *PIP* form. It consists of a list of numbered problems. Solutions and notes can be connected to problems through the identification numbers. In addition, space is allocated for *standards*, including a user's coding standard, counting standard, and the defect type standard. The defect type standard can be modified. Finally, PSP Studio provides the task *scripts* that provide on line help.

2.8. Current research in PSP.

The PSP has two major research projects in progress at this time. In both cases, only the framework of the research can be defined because the research is still in progress.

2.8.1. Team Software Process.

Watts Humphrey is now working on a process called the Team Software Process (TSP) (Humphrey, 1998). The TSP applies CMM principles and PSP methods to a team based software project. It is designed to allow a team of PSP trained engineers through the team building steps of goai management, role definition, planning, and risk management. The TSP was developed with five objectives in mind (Humphrey, 1998):

- Build self-directed teams who plan and track their work, establish goals, and own their own processes and plans. These can be pure software teams or integrated product teams of 2 to 20 PSP-trained engineers.

- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

There is inadequate data available to demonstrate the effectiveness of the TSP, however results obtained in the current testing show how the process helps teams work more effectively.

2.8.2. One Person Project.

Earlier in this chapter it was found that the PSP could not support every key practice area in the CMM. The objective of the One-Person Project process (Frankovich, 1997) is to create a process that would provide coverage for these key practice areas. The One-Person Project will be developed through supplementing the existing PSP practices with additional phases and tasks that can cover the CMM activities not already covered under PSP. The project is a thesis being developed as a joint project between the University of Calgary and Advanced Information Systems in Peoria, Illinois.

2.9. Summary:

This chapter has outlined the Personal Software Process, developed at the Software Engineering Institute by Watts Humphrey. This review included the process framework, its motivations, and how it can be used to bridge the gap between the academic environment and the software industry.

Chapter 3 provides a critical analysis of the PSP, including its structure, impact, and the automated support available for it.

Chapter 3. Critical Analysis of PSP.

In Chapter 2, it was found that the Personal Software Process (Humphrey, 1995a) provides many benefits that would help a programmer. However, it was found that the receipt of these benefits has a significant cost.

It was also noted that not every engineer would gain benefits from using the PSP as documented in the book "A Discipline for Software Engineering" (Humphrey, 1995a). However PSP provides an excellent framework for the development of a custom personal software process.

This chapter will provide a critical analysis of the Personal Software Process (PSP). The investigation and analysis covers three components. First, the advantages and disadvantages of the PSP will be investigated. An analysis will then be given on the current research into the benefits of PSP. Third, advantages and disadvantages of the four tools documented in Chapter 2 will be discussed.

Through the investigation of advantages and disadvantages, it can be supported that the need that the PSP must be customized to one's own practice. Secondly, the analysis of the current PSP automated support can be utilized towards the requirements phase to support a customized PSP software tool.

3.1. The PSP Framework.

A *software process* is a series of detailed steps that one follows in order to produce a software product. No two software processes are alike, and each has its own advantages and disadvantages. The PSP consists of a number of activities that have their strengths and weaknesses.

3.2.1. PSP Advantages.

Probably the most distinct advantage of using the PSP forms would have to be its strict standards for logging time. The Time Recording Log supplied for PSP provides the ideal template for recording time spent in the project. There are fields to log the exact starting time and ending time, making the time spent in each phase easily computed. PSP also forces one to track interruption time (Humphrey, 1995a), allowing one to determine how much of his time was actually spent working. With the comment field, one can track the causes of interruptions.

The Defect Recording Log also follows a strict standard. It makes the user account for every defect, whether it is a typographical error or a logical error found in the test phase. With each defect being assigned a unique identifier, this allows the PSP to support the multiple defect problem (Humphrey, 1995a). The multiple defect problem occurs if a second defect is found in the process of fixing a defect. Since PSP requires a repair time for each defect, allowing for an analysis of the relationship between fix time and the phase of defect removal (Humphrey, 1995a). It is generally found that the longer the life of a defect, the longer the time required fixing it.

The Process Improvement Proposal (PIP) is a powerful tool. While the form gives the impression a solution is needed immediately, it is not. However, it forces the programmer to think about it.

The Test Report Template introduced in PSP 1 is an equally powerful tool. It gives the user the leverage to determine test cases in the development phases. When it is time to perform the test phase, the test cases are ready to run. This supports the V-shaped software development model where test cases are developed as early as the design phase.

The Task and Schedule Planning features introduced in PSP 1.1 provide the user a complete set of tools for planning. The Task template forces the user to decompose the project (each PSP phase usually constitutes one task) and estimate the time for each task independently. The Schedule template allows the user to allocate tasks to each working day. Through planned and earned value project tracking is supported. It approaches tracking conservatively as no earned value is accumulated until the task is officially finished. Computing the planned and earned values based on the tasks' planned times is consistent and unbiased.

The reviews introduced in PSP 2 are an integral part of a defect awareness and prevention strategy. It should be appreciated how the PSP encourages us to adjust review checklists to accommodate common problems. The "yield" metric provides a powerful determination of the effectiveness of one's review practice.

The following metrics in PSP are found to be useful:

- Defects density (Defects per KLOC): This measure of defect density is straightforward and is a good measure of product quality and is commonly used in industry.
- Reuse Percentage: The PSP defines code as reused if the programmer wrote it in a previous project. Code from standard language libraries and templates are not considered as reused code. While engineers may have different notions on software reuse, this metric provides a significant effect on the amount of effort put in a program. The higher the reuse percentage the lower the effort.
- Defect removal leverage: Taking the ratio of defect removal rates of a review phase compared to a test phase provides an appreciation of how effective the phase serve in the removal of defects as compared to the testing phase.

3.2.2. PSP Disadvantages.

The PSP has several disadvantages that may deter people from using it.

There is one missing component in the Time Recording Log. It does not force the user to put a time stamp at the start and end of each interruption. The user simply estimates the amount of time he was interrupted. This estimate can be subjective. Therefore, it is a good idea for the engineer to implement the extra time stamps to compute interruption time.

PSP uses Logical Lines of Code as a size measurement standard. Lines of code may be easy to count, but there is no clear definition on what constitutes a line of code. There are many ways to count lines of code, including the number of physical lines, the number of method calls, the number of semi colons, or the number of non-comment lines of code. Regardless of how lines of code are counted, a consistent standard must be adopted (Humphrey, 1995a). It should be realized that the lines of code measurement is not practical for some technologies (object oriented code). It is also difficult to visualize lines of code at the early stages of a project.

The way that PSP categorizes lines of code is subject to debate. While most of the size categories are self explanatory, the concept of reuse brings the possibility of bias between engineers. PSP provides a narrow definition for reuse. It defines reused code as unmodified code from one program that is used in another. Theoretically, there is much more software reuse than we think when a software product is written. For example, we reuse classes in the `java.lang` package in every Java program. Since the sizes of external libraries are constant, they are not counted.

The code counted as reused in PSP is considered to be internally reused code. PSP does not count the reuse of standard components and libraries supplied to software engineers from external sources. However, external reuse is important. Even though externally reused code is static in size, it is impossible to count the lines of code of externally reused components. A metric that is more suitable for external reuse measurement in PSP is External Reuse Frequency (Succi and Benedicenti, 1997) which measures the proportion of total references in the code that refer to reused components that were supplied by external libraries.

The linear regression baseline for PROBE makes it a powerful estimation tool. However is ineffective if the historical data points have little or no linear correlation (scattered all over the space of a scatter diagram). While a best-fit line can be drawn for any set of data points, a high error will result in the projected estimate unless there is a statistically significant correlation.

While the design templates introduced in PSP 2.1 provide a good guideline for program design, they would be redundant for many engineers. In industry, engineers may already have external software design tools that are specific to their practice. These tools may make one or more the design templates in PSP 2.1 redundant. Tools such as Rational Rose and GDPro provide functionality to design and generate software code. In the case Rational Rose, it supports state diagrams.

The same principle applies to the PSP task and scheduling templates. Many engineers have external task planning tools like Microsoft Project and Microsoft Schedule. This would make the Task Planning and Schedule Planning templates in PSP 1.1 redundant.

The cyclical framework found in PSP 3 provides the proper framework for the incremental development of programs. However, the Cycle Summary template (Humphrey, 1995a) does not support the cycling of planning, high level design or post mortem activities. There are three reasons why these activities should also be cycled:

- A completed increment must be measured and productivity data for the increment should be accumulated.
- Adjustments in the plans for the next increment may be necessary depending on happenings during the development of the first increment.
- Some programmers may complete only a subset of the PSP activities in the same cycle. This will lead to a quantity of phase and cycle combinations having zero time on the Time Recording Log.

In addition, some components of the PSP metrics plan have weaknesses. While "Yield" is a powerful metric in measuring the effectiveness of reviews, it can be confusing. In the case of a program with few defects, a found or missed defect can seriously alter this metric. It must be taken in accordance with the product. A low yield may not always imply that many defects are slipping through the review. Another drawback is that the view of a component of code as reusable is subjective, which introduces bias into the metric of "new reusable".

Meanwhile the notion of productivity is also varied among programmers. The PSP measures productivity by dividing new and changed lines of code by the project time. This notion of productivity can be considered weak. It assumes there is no cost to reuse code, but time must be spent sizing and importing the reused components. In the same way, it also does not account for the cost of identifying lines of code that are actually removed from a base program. It is also noted that PSP seems to assume a linear relationship between size and effort, where in real life the relationship is found to be exponential as per the COCOMO model (Boehm, 1981).

In general, following the PSP exactly will not be feasible for most engineers given the technology and environment they are subject to. The PSP should be treated as a baseline for the development of quality software development practices. Each of its methods should be adjusted to one's own technology, practice, strengths, and weaknesses (Humphrey 1995b).

It should also be noted that the PSP is geared specifically towards the development of software. It lacks the budgeting of time that is spent eliciting requirements. PSP assumes that a programmer knows the requirements for the software that he is producing. Eliciting requirements is a key task that could be considered a phase in PSP. The same case can be said for post-development activities such as documentation.

3.3. Customizing the PSP to a Personal level.

Humphrey says, "defined personal processes should conveniently fit the individual skills and preferences of each software engineer" (Humphrey, 1995a). With any change, execution of the PSP on paper may require significant adjustments to the forms as the change provides a ripple effect through the entire process. Changes to the PSP can happen in four ways.

3.3.1. The customization of phases.

The PSP provides eight basic phases for software development (Humphrey, 1995a). These phases cover the basic tasks for developing a software product. However, these eight phases may not be adequate for larger scale projects. The following cases provide examples that would require a change to the basic phase list.

- The individual software engineer may wish to add a phase specifically for the non-production stages such as risk management, requirements negotiation and formal specification.
- In a development environment like IBM's Visual Age for Java, the code is automatically compiled as it is saved. This makes the compiling phase unnecessary.
- Design phases can be divided into several levels of design as in PSP 3 (high level and detailed level design).

3.3.2. The customization of defects.

The PSP provides a defect type standard. PSP uses defect types to sort defects by their causes. Each user has his or her view of what a defect is. A user may develop additional defect types that reflect his practice. The user may also sub-type defect types to reflect his most common problems. In fact, some programmers do not recognize syntax errors as defects.

3.3.3. The customization of software size measurement.

Not all developers prefer to use lines of code for initial estimates. Some would prefer to use function points, the number of object classes, or the number of requirements as the unit of measurement. In addition, the final software size may be measured in a different unit than that of the estimate. Effective use of this strategy would require a mathematical model that links the two distinct measurement units together.

Also, not all developers wish to measure their performance against "new and changed" lines of code. Some may prefer to use total LOC, or may use a totally different sizing method.

3.3.4. The customization of checklists.

As in the case of defects, each user has his or her preferences for what points must a software design document or code block be inspected against. Most users wish to base their checklists on the mistakes they have done in the past.

A problem that is commonplace can be added as a check item. Similarly, a check item that has become a non-issue for a programmer could be removed from the checklist.

3.4. Research on PSP Benefits.

As was mentioned in Chapter 2, Humphrey noted many benefits of using the PSP. However, in order for these benefits to be proven true, they have to be identified through user studies. This section evaluates the effectiveness and validity of the research that has been conducted with the PSP. Two types of research were evaluated: the effects of taking the PSP course and the effects of implementing PSP methods into the undergraduate computer science program starting in the first year.

3.4.1. The effectiveness of the PSP course.

The following pattern is noticed whenever a statistical study of the impact of PSP is undertaken.

- The estimation process was found to be statistically unstable until level 1. When the PROBE method was introduced, estimations became more consistent.
- The pre compile defect yield was found to be very low until code and design reviews were introduced in level 2.

This pattern is simply to be expected because of the incremental nature of how PSP is taught in the course (Humphrey, 1995a). Humphrey could have dealt the entire set of PSP methods at the same time, however the seven level incremental approach used in the course (Humphrey, 1995a) makes the most sense. If all the methods were introduced at the same time, students would feel overwhelmed by the sudden increase in the workload. The incremental approach allows an engineer to make a smooth transition of improving his personal process by learning one method at a time.

While PROBE appeared to help the estimation process at the course level, it may not be practical when it is applied for a large program. Lederer and Prasad conducted an investigation into estimation error for different size estimation bases (Lederer and Prasad, 1998). They concluded that the use of algorithmic estimation methods like PROBE does not of itself increase estimation accuracy. It was found that such methods are not guaranteed to improve accuracy. A key factor may be that algorithmic methods do not provide adequate coverage of key influences (e. g. change) in setting estimates. Researchers require further understanding of how a particular method can improve estimation accuracy.

The most interesting find by Lederer and Prasad was that inaccuracy in estimates was reduced only when the estimators were held accountable. Therefore, estimation accuracy should be included in a performance review and personnel who produce consistently accurate estimates should be suitably rewarded.

3.4.2. Introduction of PSP methods into the undergraduate program.

Reports were published on two experiences involving introduction of the PSP methods in the first year of an undergraduate program. Embry Riddle Aeronautical University in Daytona Beach, Florida (Hilburn and Towhidnejad, 1997) and the University of Utah (Williams, 1997) initially used the same strategy in introducing the methods into their first year courses. Time management, scheduling, and planning would be taught as part of the first half course. Defect management, and size estimation would be taught as part of the second half course.

Both universities made mistakes in administering time management in the first half course. They administered the course in a way such that the students were not encouraged to do the tasks. In these cases, time management and logging is not the priority of freshman students. The concern of the first year student is getting assignments done as well as learning to use the technology supplied to them. Since these students find little reward in the short term from logging their time usage, they find the task a liability.

The institutions learned from this mistake and adjusted their strategies such that students would realize the benefits. For example, the University of Utah successfully moved defect logging to the first half course while moving time logging to the second half course (Williams, 1997). Defect logging and prevention would raise a higher interest from students. Students expect to be removing defects during programming assignments. The defect management aspect of PSP would make students excited since they would encounter less frustration in compiling and testing.

If PSP activities are applied in a first year course, they should be fully covered within the first few lectures of the semester, such that data collection is done right away. However, to reduce the already heavy course load faced by first year students, PSP time logging should be done at a high level. The framework of course activities should be timed as opposed to every discrete task. (Hilburn and Towhidnejad, 1997).

To encourage students to do the tasks, the actual PSP tasks must count towards the student's grade. While the PSP related portion of the grade should be small, it should be large enough such that doing the activities makes the difference between A and A-.

Williams (1997), Hilburn (1997), and Bagert (1997) recommended the development of automated support to support the PSP activities. Such automated support should be easy to use and able to run on the same technology in which programming is implemented.

The text "Introduction to the Personal Software Process" (Humphrey, 1997) serves as an ideal companion text for the first year courses. It provides a detailed foundation of the PSP methods and the reason they are there. While cyclical development is not covered in the text (Williams, 1997), the guidelines for such development are there.

3.5. PSP Automated Support.

There are four PSP support tools currently available. In evaluating the effectiveness of each tool, the following questions will be covered:

- Would this tool save time in executing PSP?
- Would it be transparent to the task at hand?
- What degree of user customization is supported?
- What components of PSP does this tool cover?

3.5.1. psptool0.6.

Andrew Wolesley created this TCL-TK script to implement the PSP time and defect tracking (Wolesley, 1997).

This tool would save time in executing the time and defect collection portions of the PSP. First, the "Run-Pause" button combined with automatic creation of Time recording log entries makes time recording simple. Secondly, the "bug" button allows defects to be brought up at any time. Defect fix times are accumulated automatically as the timer runs. The tool supports end user customization through the addition of new defect types. It is platform independent within the limits of the TCL-TK language.

While this tool has benefits, there are many drawbacks that make this tool undesirable for use. The first main problem is only a subset of the PSP activities are supported. Critical components of the PSP like checklists are not supported even though review phases are included in the phase list. Another problem is that interruptions are not explicitly timed. It is important to know the amount of time lost to interruptions.

The time estimation component could use some attention. All planning estimates are based on the software size estimate and the "To date" percentage fields from previous programs. The user cannot manually adjust these estimates for a problem outside of his domain knowledge.

A critical problem with psptool is the high cost of making a mistake. First, there is no facility to edit incorrect time and defect entries. They can be edited only through an explicit opening and manipulation of the project file. Second, finished projects cannot be reopened. Finally, if a new project is created under an old project name the old project files are lost.

3.5.2. Timelog.

As specified by its name, Timelog (Clemens, 1997) specifically implements the PSP Time Recording Log. It does an excellent job. This tool has potential to serve as a base for expansion into the other PSP data collection areas.

The tool has a number of excellent benefits. First, it is platform independent within the limitations of Java. Second, it can support multiple processes. PSP0 is automatically created when the system is installed. New processes (lists of phase names) can be created. As new projects are created, the user is asked to choose the process he can use. In addition, Timelog supports the editing of any entry in the log.

However, Timelog has the same problem with psptool as it only supports the Time Recording Log component of the PSP. It does not support any other PSP related task such as defect logging.

The edit facility would be improved if it were cleaned up when the entry is added to the Time Recording Log. This may confuse the user. Also, making a mistake in editing a time entry can be frustrating. If a user manually edits a time field and breaches the format, the old time is restored.

The author carries one more comment on Timelog. If the start time is "ahead of" the stop time, the tool automatically assumes that midnight has passed when computing a delta time. He is assuming that the user will not leave the application running over several days.

3.5.3. Timmie.

The use of Timmie (Institute for Program Structures and Data Organization, 1997) is simple. All that has to be done is highlighting the current phase of the work. Time is automatically accumulated in that phase. Events for each of the phases can be logged by double clicking on the phase name. Time stamps can be added allowing the user to add comments and log defects. Interruptions can be logged, however, the event window for that project must be open for the interruption to show on the event log. Otherwise the interruption is ignored.

Timmie was not specifically designed to support PSP. Like Timelog, Timmie does not support any other component of the PSP. Only a brief time summary may be generated through the use of the Perl script "accumtimmie". This script uses a command line interface whose parameters are not easily remembered.

Even though defects can be logged through the event window, there is no formal mechanism or script that can be used to generate a formal PSP Time or Defect Recording Log.

Timmie does not support printing, however the format of the text based project files can be dumped to a printer.

3.5.4. PSP Studio.

The current version of PSP Studio (Henry, 1997) is a very promising tool that supports the Software Engineering Institute's specification for PSP automated support. PSP Studio supports all seven levels of the PSP, along with its forms and templates. Edit facilities exist for every form.

This project has been geared the instructors and students who are involved in the PSP training course. It would be impractical for industry to use it. A major problem is that it will run only in the Windows environment, and many firms do not use a PC based environment.

The author found it very annoying that the design and code review checklists started as empty. In one way, this is OK as the user may define a checklist based on his specific problems. On the other hand, having a standard set of items in these checklists would provide the user help in defining his checklists.

There are a couple of cosmetic problems that force the user to go out of the way to do simple tasks. First, PSP Studio forces the user to move to the appropriate form to perform an action on it. While a toolbar is available, only basic file functions are supported. It is essential to expand the toolbar to support common PSP tasks such as timing. The second problem is the layout of the forms. The forms are organized using two levels of tabs, which the author found difficult to navigate. The forms would be better organized in a tree layout.

3.6. Summary.

This report has outlined the advantages and shortcomings of the Personal Software Process developed by Watts Humphrey. The report included the advantages and shortcomings of the PSP framework, the research into the PSP, and a list of tools used to automate PSP data collection.

The review of the PSP and the currently available software tools will provide help in the requirements elicitation in Chapter 4, as well as the design (Chapter 5) and implementation (Chapter 6) of automated support for the user-changeable PSP.

Chapter 4: Requirements Analysis.

This chapter is dedicated to the elicitation of the requirements for the PSP support tool. This tool shall be known as PSP Manager. It has been decided to examine use cases in this investigation.

The requirements document starts off with a number of assumptions and definitions. In developing requirements, each component of the PSP will be investigated separately. A series of use cases will be converted into a list of requirements for that component to be successfully implemented. After each component has been investigated, additional general requirements will be developed such that the tool requirements are complete.

Since customization of the tool to one's personal process is a major objective of this project, emphasis will be placed on identifying use cases and requirements that will help in the fulfillment of this objective.

For the purpose of this requirements elicitation, the term *publish* refers to converting PSP data into a user readable format that can be printed.

4.1. Assumptions.

The following assumptions have been made in the elicitation of the requirements.

- It is assumed that the users of the PSP Manager have an understanding of PSP.
- It is assumed that external software development tools are available for the purpose of software design and implementation.
- It is assumed that external tools are available for the purpose of task and schedule management.
- It is assumed that the programmer has access to a software tool for the purpose of software size measurement and that the result will be entered as the official size metric of the project.

- It is assumed that a task cannot be interrupted unless it is in progress.
- It is assumed that each problem in a PIP form is associated with one solution.
- It is assumed that the time span for one entry in the Time Recording Log will be no longer than 48 hours.
- It is assumed Microsoft Excel is the external spreadsheet package.
- It is assumed that the programmer has the facilities to perform a backup and restoration of the programmer profile.
- It is assumed that the programmer will use the security features of his computer or network to protect his user data.
- It is assumed that only one project will be worked on at a time and that users may switch between projects through the normal open and close operations.

4.2. Personal Process Design (Phases).

A phase is defined as a structured task of the software process (Humphrey, 1995a). A process consists of several phases with each phase having a unique name. The PSP phases are structured in an order that is natural in the development process. However, the tool should be structured such that the user can run in any phase he feels comfortable with (SEI, 1998).

The Software Engineering Institute recommends that a user be able to rename the development phases (SEI, 1998). It is also beneficial for a user if he could add, delete, or reorder the process phases to define a customized process.

To implement PSP at level 3, the phases are repeated in order a specified quantity of times. Each iteration of the phase list constitutes one PSP cycle (Humphrey, 1995a). The cycles are numbered in order from one to the number of cycles planned.

4.2.1. Use Cases for Phase Definition.

The user can customize the phase list to his process by adding, renaming, deleting, and reordering the phases. Since the ordering of the PSP phases is important, the phase names must be stored in a list that emphasizes order. The user interface that allows the user to manipulate the phase list must account for the prcper order. This interface can be requested at any time. Phase definition requires the four use cases in Figure 2.

4.2.1.1. Use Case: User defines new phase.

If a phase is to be added to the list, the user first requests for the list of phases. He then requests that a new phase be added under the name identified by him. An error condition occurs if the name of the new phase already exists in the list.

4.2.1.2. Use Case: User no longer needs existing phase.

If a phase is no longer needed, the user requests the list of phases. He then selects the phase and request that it be deleted from the list.

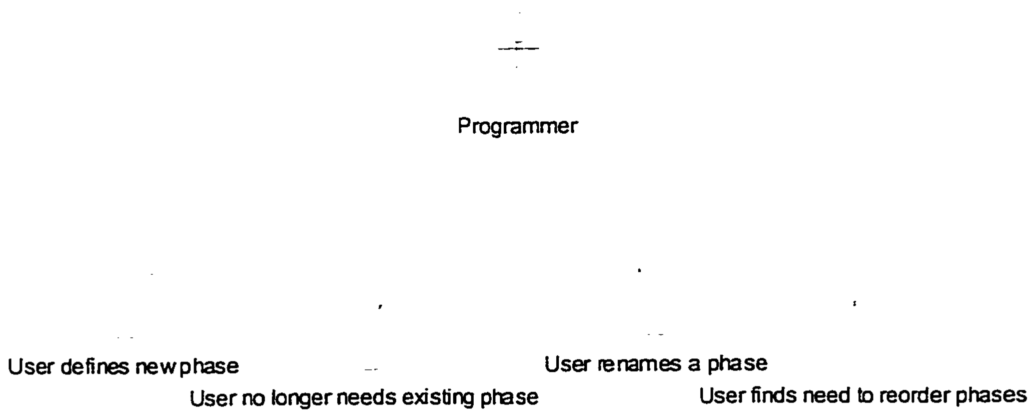

Programmer
User defines new phase
User no longer needs existing phase
User renames a phase
User finds need to reorder phases

Figure 2. Use case diagram for phase management.

4.2.1.3. Use Case: User renames a Phase.

To rename a phase, the user selects the phase to be renamed and types in the new name. The new name of the phase must not exist elsewhere in the list.

4.2.1.4. Use Case: User finds need to reorder phases.

The ordering of the phase list implies the order in which they will be done. While accessing the user's phase list, the user may select a phase and request that it be moved to a new position in the list. The first phase in the list cannot be moved earlier in the order and similarly the last phase cannot be moved later in the order.

4.2.2. The requirements for Phase Management.

Based on these four use cases, the following requirements are derived:

Requirement 4.2.2.1. PSP Manager shall maintain an ordered list of PSP phases.

Requirement 4.2.2.2. PSP Manager shall ensure the phases in a phase list are uniquely named.

Requirement 4.2.2.3. PSP Manager shall provide a user interface that lists the available PSP phases in the order of completion.

Requirement 4.2.2.4. PSP Manager shall allow the user to add a phase to the list.

Requirement 4.2.2.5. PSP Manager shall allow the user to delete a phase from the list.

Requirement 4.2.2.6. PSP Manager shall allow the user to rename a phase in the list.

Requirement 4.2.2.7. PSP Manager shall allow the user to move a phase towards the front of the list.

Requirement 4.2.2.8. PSP Manager shall allow the user to move a phase towards the end of the list.

4.3. Timing.

Time management is the most important component of data collection in the software development process. It is the most frequently performed activity in the process. The PSP script states that every use of time on a project phase must be logged. PSP also requires that interruption time be tracked. If disruption time is not tracked, the user does not get an accurate use of time actually spent in the phase (Humphrey, 1995a).

4.3.1. Use Cases for the Timer.

Five use cases are required for the creation of time recording log entries as shown in Figure 3.

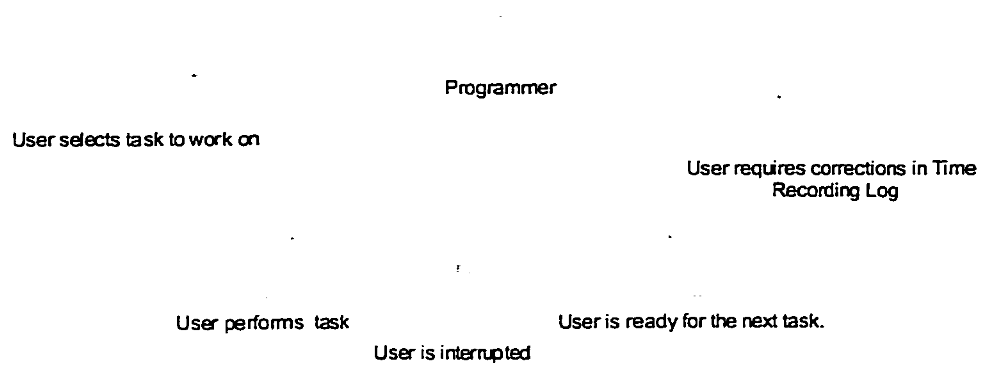

Programmer
User selects task to work on
User requires corrections in Time Recording Log
User performs task
User is ready for the next task.
User is interrupted

Figure 3. Use case diagram for timing.

4.3.1.1. Use Case: User selects task to work on.

The user may request the timer to operate in the phase of his or her choice. The user selects the phase and requests the timer to switch to that phase. If the task is changed when the timer is running, an entry is created for the previous task and the timer continues running.

4.3.1.2. Use Case: User performs a task.

Before starting the task, the user selects the appropriate PSP phase and cycle number to operate in and starts the timer. The user proceeds with his task. If a worker should be interrupted before he can start the timer, a facility is available to edit the time recording log entry such that the disruption can be accounted for. The user stops the timer when his task is done. A new entry is added to the time recording log based on the amount of time spent in the phase

4.3.1.3. Use Case: User is interrupted.

Many external factors such as phone calls and unexpected company can cause work to be interrupted. The user stops the timer when this happens. When the interruption is over, the user requests the timer to resume. The amount of disruption time is computed and added to the accumulated disruption time for the current task.

4.3.1.4. Use Case: User is ready for the next task.

When the user is done a task and wants to carry on to the next, he can request the timer to set the next phase as current. An error condition occurs if the current phase is the last phase of the last cycle.

4.3.1.5. Use Case: User requires corrections in Time Recording Log.

The user may request the Time Recording Log at any time in the project such that corrections can be made. An entry may be corrected by selecting that entry and asking that it be edited. If an extraneous entry is in the Time Recording Log, the user may select it and request its deletion.

4.3.2. Requirements for Timing.

Based on the use cases for timing, the timer for the PSP Manager must fulfill the following requirements.

Requirement 4.3.2.1. The PSP Manager timer shall automatically create the Time Recording Log for the current project.

Requirement 4.3.2.2. Each Time Recording Log entry shall consist of a starting time, an ending time, an amount of time lost to interruption, the actual amount of time spent in the task, the appropriate phase name, the cycle number and a user comment.

Requirement 4.3.2.3. The PSP Manager timer shall support the accumulation of interruption time over multiple interruptions in the same task.

Requirement 4.3.2.4. PSP Manager shall allow the user to access to the project's Time Recording Log at any time in the project.

Requirement 4.3.2.5. PSP Manager shall allow the user to edit entries in the Time Recording Log.

Requirement 4.3.2.6. PSP Manager shall allow the user to delete entries in the Time Recording Log.

4.4. Defects.

Defect management requires the management of a list of general defect types and a list of project specific defects (Defect Recording Log).

4.4.1. Use Cases for Defect Types.

The User shall be automatically supplied with the ten defect types defined by Humphrey. A user may customize a defect type list to his process by either defining a new defect type or through deriving a subtype of an existing defect type. The four use cases in Figure 4 are required to manage the list of defect types.

4.4.1.1. Use Case: User identifies new defect type.

The user can access the list of defect types and request that a new type be added to the list. An identification string for the new defect type is automatically created and the user enters a description of the defect.

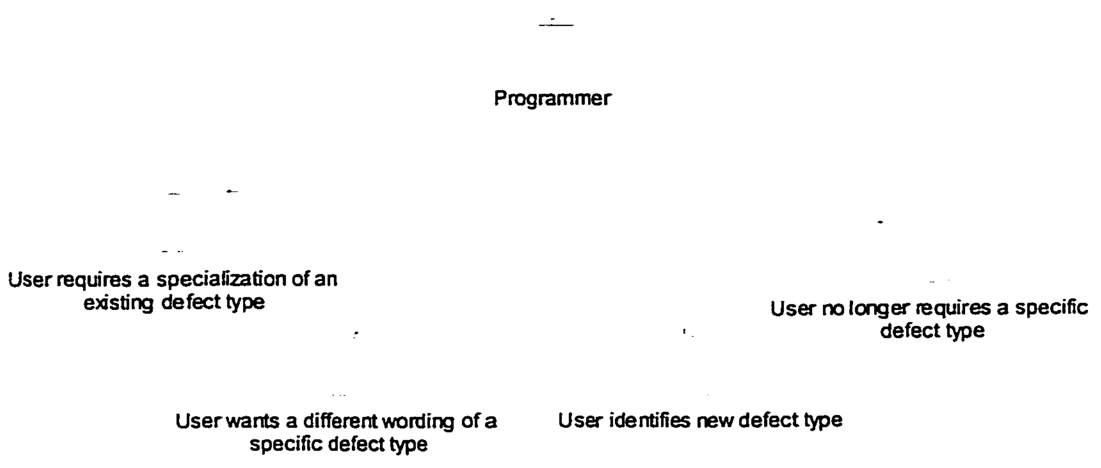

Programmer
User requires a specialization of an existing defect type
User no longer requires a specific defect type
User wants a different wording of a specific defect type
User identifies new defect type

Figure 4. Use case diagram for defect type management.

4.4.1.2. Use Case: User requires a specialization of an existing defect type.

Sometimes a user has a problem with a certain defect that is related to an existing defect type. In this case, the user can select the problematic defect type and request a new defect type as its sub-type. The identification number of the new defect type is created from its super-type while the user enters its description. Sub-types of defects can be sub-typed many levels.

4.4.1.3. Use Case: User wants a different wording of a specific defect type.

The user may select any defect type and request it to be edited. This allows the user to change the wording such that he has a better understanding of a defect type.

4.4.1.4. Use Case: User no longer requires a specific defect type.

If a defect type is not a problem, the user may select that defect type and request it to be deleted.

4.4.2. Requirements for Defect Type Management.

The four use cases for defect type management lead to five requirements.

Requirement 4.4.2.1. PSP Manager shall provide functionality to allow the user access to a user's defect types at any time.

Requirement 4.4.2.2. PSP Manager shall provide functionality to allow the user to delete defect types.

Requirement 4.4.2.3. PSP Manager shall provide functionality to allow the user to rename defect types.

Requirement 4.4.2.4. PSP Manager shall provide functionality to allow the user to create a new independent defect type.

Requirement 4.4.2.5. PSP Manager shall provide functionality to allow the user to create a subtype of an existing defect type.

4.4.3. Use Cases for the Defect Recording Log.

There are three use cases in the development of the Defect Recording Log as documented in Figure 5.

4.4.3.1. Use Case: User finds defect.

When a new defect is found, the user requests to see the Defect Recording Log. He then asks for a new defect to be defined. The program responds by providing the edit facility such that the user can edit it. The user attempts to fix the defect. When the defect is fixed, he marks the defect as fixed. If the defect cannot be immediately fixed, he can mark it as unfixed and reopen it later on. The time to fix the defect is automatically recorded.

4.4.3.2. Use Case: User edits defect.

The user may select any defect from the Defect Recording Log and request it to be edited. If the defect is still outstanding, the user should attempt to fix it and the fix time for the defect will accumulate.

4.4.3.3 Use Case: User adds a defect in error.

If a defect was erroneously introduced, the user may select it and request it to be deleted from the log.

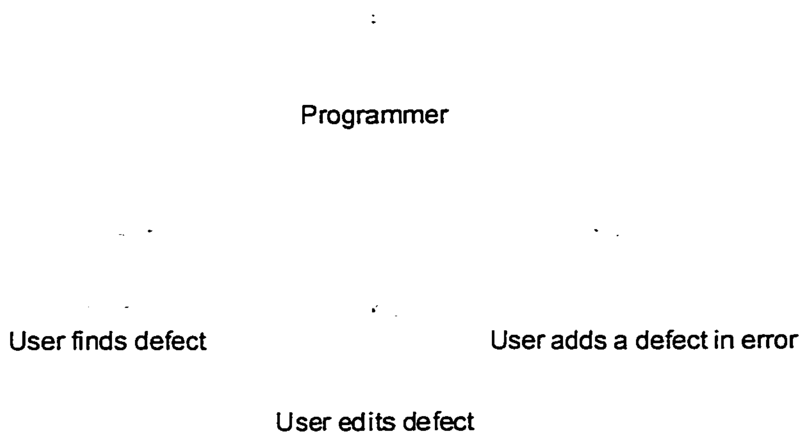


Figure 5. Use Case Diagram for Defect Recording Log.

4.4.4. Requirements for Defect Management.

The four use cases define six requirements to complete the implementation of the Defect Recording Log.

Requirement 4.4.4.1. PSP Manager shall provide functionality to allow the user access to the project's Defect Recording Log at any time in the project.

Requirement 4.4.4.2. Each defect shall consist of an identification number, a date in which it was found, a phase of injection, a phase of removal, a defect type from the programmer's list of defect types, a fix time, and a field for a programmer comment.

Requirement 4.4.4.3. PSP Manager shall allow the user to add a defect to the Defect Recording Log.

Requirement 4.4.4.4. PSP Manager shall allow the user to delete a defect from the Defect Recording Log.

Requirement 4.4.4.5. PSP Manager shall allow the user to edit a defect in the Defect Recording Log.

Requirement 4.4.4.6. The edit facility for defects shall allow the user to mark a defect as fixed, save a defect details without fixing, or discard the changes

4.5. Metrics.

The metrics of PSP can be divided into three categories:

- *User defined raw metrics* have names and values explicitly defined by the user. The most important metric whose source is the user is software size.
- *User defined computed metrics* have names explicitly defined by the user. Their values must be computed from existing PSP data.
- *Cumulative metrics* have a phase name associated with their values. They are computed through queries to the time and defect recording logs.

A project requires both a planned value and actual value for each user-defined metric. PSP data and formulae for user defined metrics shall be collected from the user by PSP Manager. At the request of the user, the Manager will produce a text file that can be converted to a spreadsheet in Microsoft Excel.

In allowing the user to support a custom personal process, such user shall be permitted to make changes to the metrics plan provided by PSP such that it complies with his practice.

4.5.1. Use Cases for PSP Metrics.

The PSP Manager requires the management three categories of metrics (project plan, project actual and career "to date"). Since the definition of software size is user dependent, it is placed in the list of user metrics. One of the metrics in the user metric list is marked as the "official size metric".

In Figure 6, six use cases are defined to manage the list of user metrics.

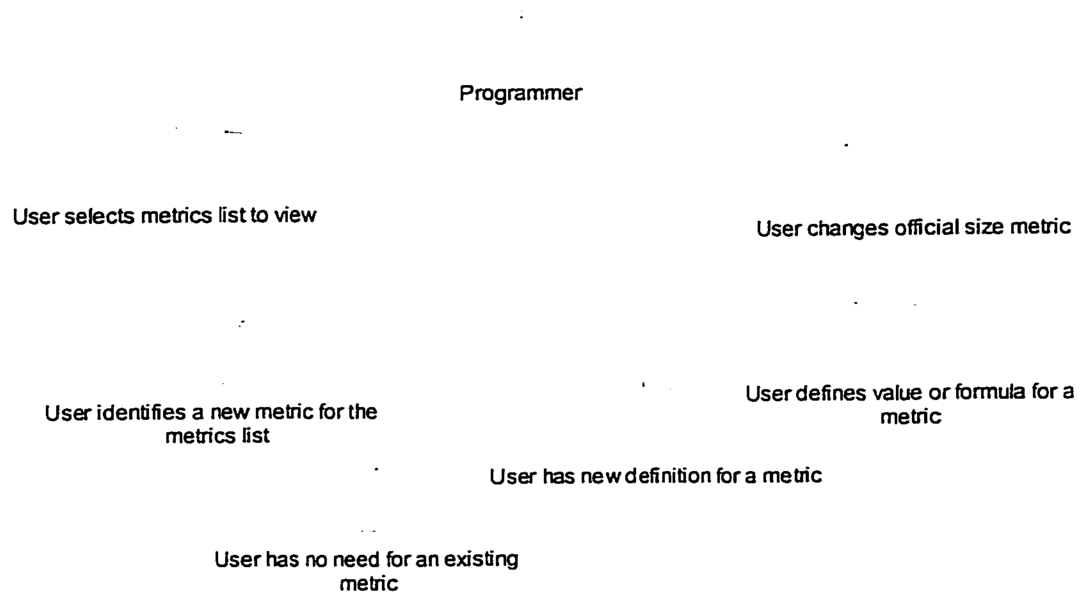

Programmer
User selects metrics list to view
User changes official size metric
User identifies a new metric for the metrics list
User defines value or formula for a metric
User has new definition for a metric
User has no need for an existing metric

Figure 6. Use case diagram for the user metric list.

4.5.1.1. Use Case: User selects metrics list for viewing.

The user has access to three metrics lists. The master list is the list of metrics automatically assigned to the new projects as they are created. There are two project specific metrics lists: one for the estimates and one for the actual metrics list.

4.5.1.2. Use Case: User identifies new metric for the metrics list.

The user may request a new metric in the metrics list. It may be added at either a project or a process level. The program opens the edit facility to allow the user to assign the new metric a name and data type.

4.5.1.3. Use Case: User has no need for an existing metric.

The user can select a metric from a metrics list and request its deletion. The official size metric cannot be deleted. It may be necessary to update the spreadsheet formulae to accommodate this change.

4.5.1.4. Use Case: User has new definition for a metric.

The user may select a metric in the list and request its specifications be edited. In editing a metric specification, one can modify the metric's name, its measurement unit, its type (time, real or integer) and its source (user supplied or spreadsheet formula).

4.5.1.5. Use Case: User sets value or formula for metric.

The user may select a metric in the list and request its value be edited. Either the metric's formula or value can be edited depending on its type. The current value of the metric is presented in the proper format for the type of data that it stores.

4.6.1.6. Use Case: User changes official size metric.

The user can select a metric in the list and request it to be marked as the "official size metric". It may be necessary to change some spreadsheet formulae in the other metrics to accommodate this change.

4.5.2. Requirements for User Metric Management.

The six use cases defined in Section 4.5.2 lead to eleven requirements for implementing the user managed metrics lists of PSP.

Requirement 4.5.2.1. PSP Manager shall provide access to the overall user metric list at all times.

Requirement 4.5.2.2. PSP Manager shall provide access to the project specific list of estimated user metric values while the current project is open.

Requirement 4.5.2.3. PSP Manager shall provide access to the project specific list of actual user metric values while the current project is open.

Requirement 4.5.2.4. PSP Manager shall implement metrics that support time span, integer, and real number data.

Requirement 4.5.2.5. PSP Manager shall provide functionality to allow the user to change the name and type of a user defined metric.

Requirement 4.5.2.6. PSP Manager shall provide functionality to allow the user to change the value of a user defined metric.

Requirement 4.5.2.7. PSP Manager shall automatically create the data based metrics as they are required.

Requirement 4.5.2.8. PSP Manager shall publish the user metric list with the data based metrics as a text file that can be understood as a spreadsheet by Microsoft Excel.

Requirement 4.5.2.9. PSP Manager shall mark one of the user metrics as the official size metric.

Requirement 4.5.2.10. PSP Manager shall provide the user the ability to switch the official size metric.

Requirement 4.5.2.11. PSP Manager shall not allow the official size metric to be deleted.

4.6. Issues.

Responsible design requires the documentation and the resolution of issues. Issues are managed at a project level. Each project may create zero, one or many issues.

The Issue Recording Log consists of a list of issues and belongs to one project.

4.6.1. Use Cases for Issues.

Figure 7 documents four use cases required to manage the Issue Recording Log.

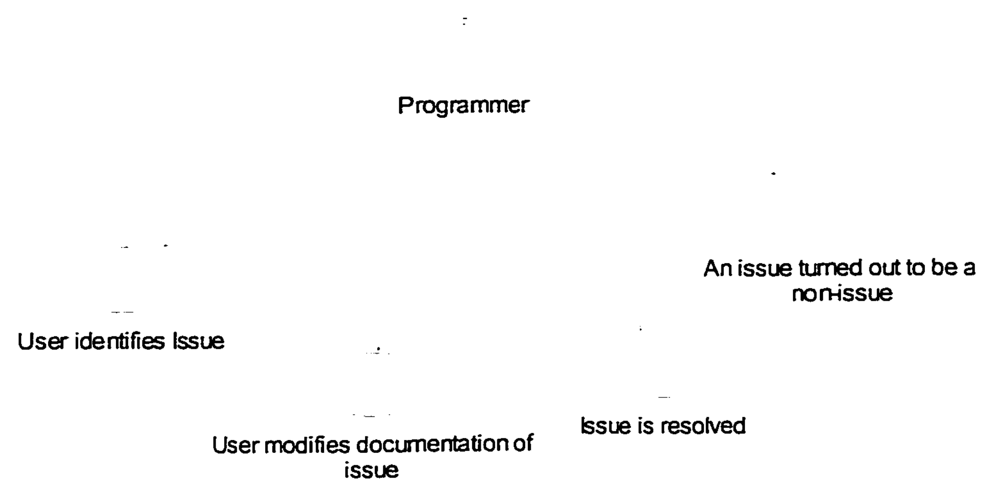

Programmer
An issue turned out to be a non-issue
User identifies Issue
Issue is resolved
User modifies documentation of issue

Figure 7. Use case diagram for Issue management.

4.6.1.1. Use Case: User identifies issue.

If a project level issue is found, the user must request the Issue Recording Log to record it. He or she then requests a new issue to be added to the log. When the edit facility opens on the new issue, the user enters its documentation. It is normal that a new issue is not resolved immediately, so the user can file it by marking it as unresolved.

4.6.1.2 Use Case: Issue is resolved.

When an issue is resolved, the user requests the Issue Recording Log, selects the affected issue and edits it. Documentation is added on how the issue was resolved before the user marks it as resolved.

4.6.1.3 Use Case: User modifies documentation of issue.

The user can reopen an issue at any time by requesting the Issue Recording Log, selecting an issue and editing it, regardless of whether the issue is resolved.

4.6.1.4. Use Case: An issue turned out to be a non-issue.

The user may request the Issue Recording Log, select an issue that was not really an issue, and request it be deleted.

4.6.2. Requirements for the Issue Recording Log.

Six requirements for the Issue Recording Log are elicited.

Requirement 4.6.1.1. PSP Manager shall provide functionality to allow the user access to the project's Issue Recording Log at any time in the project.

Requirement 4.6.1.2. An issue shall consist of an identification number, the date it was found, a phase in which it was found, a problem description, a date in which the issue was resolved, a description of the resolution, and the phase in which the issue was resolved.

Requirement 4.6.1.3. PSP Manager shall allow the user to add an issue to the Issue Recording Log.

Requirement 4.6.1.4. PSP Manager shall allow the user to delete an issue from the Issue Recording Log.

Requirement 4.6.1.5. PSP Manager shall allow the user to edit an issue in the Issue Recording Log.

Requirement 4.6.1.6. PSP Manager shall allow the user to mark an issue as either resolved or outstanding.

4.7. Reviews.

PSP involves two types of reviews: a design review and a code review. However, the work products produced by any phase may be subject to a review. A review is a formal process complete with checklists (Humphrey, 1989).

In PSP, each review phase has a different checklist with different items. Automated support for a custom personal process should allow the user to create a review checklist and associate it with any phase. PSP supplies a design review checklist and a code review checklist (Humphrey, 1995a), however each user can customize his checklist to his own practice.

4.7.1. Use Cases for Checklists.

Each phase is associated with one checklist. The ability to alter the checklists will allow the user to customize the checklists to frequent problems with his process. New (empty) checklists are added as new phases are introduced. Two use cases are documented as per Figure 8.

Programmer
User modifies a checklist
The use of a checklist in a review.

Figure 8. Use case diagram for Checklists.

4.7.1.1. Use Case: User modifies a checklist.

Checklists can be modified through the addition, deletion, and renaming of check items. Before the user can modify a checklist, he must supply the phase name that represents the checklist that he wants.

When a new criterion is identified, it can be converted to a check item by requesting a new item to be added to it, followed by entering the description of the new check item. Similarly, the user can select a check item in any checklist and edit it such that a new description of the check item can be entered. If a check item in a particular checklist is no longer useful, the user can select that check item and request its deletion.

4.7.1.2. Use Case: The use of a checklist in a review.

At review time, the user selects the checklist to be used and requests that it be published. The published checklist is opened in a window, which is used to conduct the review.

4.7.2. Requirements for Checklists.

For the implementation of checklists, six requirements were elicited.

Requirement 4.7.2.1. PSP Manager shall provide functionality to allow the user access to the user's checklists at any time.

Requirement 4.7.2.2. PSP Manager shall ensure a checklist can be maintained for every phase in the user's personal process.

Requirement 4.7.2.3. PSP Manager shall allow the user to add a check item to a checklist.

Requirement 4.7.2.4. PSP Manager shall allow the user to delete a check item from a checklist.

Requirement 4.7.2.5. PSP Manager shall allow the user to edit a check item in a checklist.

Requirement 4.7.2.6. PSP Manager shall allow the publishing of a checklist for its use in an inspection.

4.8. Test Cases.

Each project has infinitely many test cases, so a product cannot be fully tested. Each project requires a set of standard test cases.

4.8.1. Use Cases for Test Report Template.

Shown as Figure 9 are three use cases required for Test Case management.

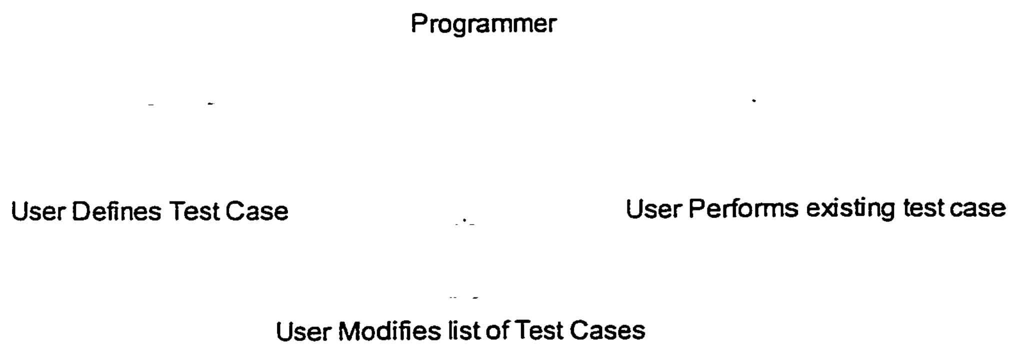

Programmer
User Defines Test Case
User Performs existing test case
User Modifies list of Test Cases

Figure 9. Use case diagram for Test Case Management.

4.8.1.1. Use Case: User Defines Test Case.

When a test case is identified, the user requests the log of test cases and requests for a new test case to be added to the list. He defines the test case with inputs and expected results. If the test case has been executed, the results of the test are recorded and the test is marked as successful or unsuccessful.

4.8.1.2. Use Case: User Modifies List of Test Cases.

The user may modify a test case by selecting the affected test case and requesting an edit on it. If a test case is found to be meaningless, the user may select that test case and request it to be deleted.

4.8.1.3. Use Case: User Performs Existing Test Case.

The user can executes a previously incomplete test case by selecting the affected case and requesting it to be edited. When the test is complete, he fills in the actual outputs and marks the test as successful or unsuccessful.

4.8.2. Requirements for Test Case Management.

Five requirements were elicited to implement management of the test reports.

Requirement 4.8.2.1. PSP Manager shall provide functionality to allow the user access to the test cases of the current project at any time while the project is open.

Requirement 4.8.2.2. Each test case shall consist of a description or objective, a list of inputs, a list of expected outputs, a list of actual outputs, and a state to determine whether the test is pending, successful, or not successful.

Requirement 4.8.2.3. PSP Manager shall allow the user to add a test case to a project.

Requirement 4.8.2.4. PSP Manager shall allow the user to delete a test case from a project.

Requirement 4.8.2.5. PSP Manager shall allow the user to edit a test case in a project.

4.9 Projects.

A project is defined as a program that is written using PSP. Each project has a name, a user assigned identification string, a description, and an owner. Each project contains the PSP data pertaining to that program. A programmer may have many outstanding projects but works on one project at a time.

4.9.1. Use Cases for the PSP Project.

Five use cases are defined as shown in Figure 10.

4.9.1.1. Use Case: A new project is started.

The user can request a new project by opening the list of projects and requesting a new project be started. The new is built based on the process defined in the programmer profile.

4.9.1.2. Use Case: User performs work on a project.

The user starts work on a project by selecting the affected project and opening it as the current project. The timer is adjusted to work for the phase list of this project. When work is finished on the project, the project is closed. When the project is closed, the timer must be checked to assure the last time recording log entry is recorded.

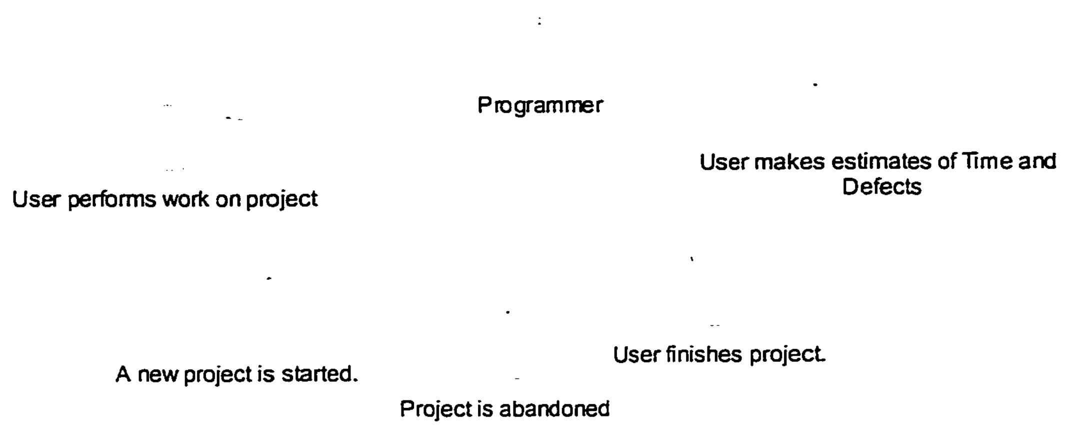

Programmer
User makes estimates of Time and Defects
User performs work on project
User finishes project.
A new project is started.
Project is abandoned

Figure 10. Use case diagram for PSP Project management

4.9.1.3. Use Case: Project is completed.

When the program is completed, the user assures all user supplied metrics are entered before marking the project as "finished". The project is moved into the programmer's historical database.

4.9.1.4. Use Case: User makes estimates of time and defects.

The user can request the Plan Summary such that he can enter time and defect estimates. These estimates are sorted by phase. The current values of the actual metrics being estimated are made available such that a comparison can be made.

4.9.1.5. Use Case: Project is abandoned.

If a project is discontinued, the user may request its deletion from the list of outstanding projects. If the user wants the project work to count in his historical database, he proceeds as if he was finishing the project.

4.9.2. Requirements for the PSP Project.

Ten requirements are elicited for the storage of PSP data for one project.

Requirement 4.9.2.1. The Project shall contain a Time Recording Log.

Requirement 4.9.2.2. The Project shall contain a Defect Recording Log.

Requirement 4.9.2.3. The Project shall contain an Issue Recording Log.

Requirement 4.9.2.4. The Project shall contain a list of Test Cases.

Requirement 4.9.2.5. The Project shall contain estimated and actual values for all metrics.

Requirement 4.9.2.6. PSP Manager shall provide functionality to allow the user access to the Project Plan Summary the current project at any time while the project is open.

Requirement 4.9.2.7. The Plan Summary actual time and defect statistics shall be built from the contents of the PSP Project logs and user metrics.

Requirement 4.9.2.8. PSP Manager shall provide current up to date values for the actual elapsed time and defects encountered in each phase.

Requirement 4.9.2.9. The Project shall not be added to a user's historical data until it is declared finished.

Requirement 4.9.2.10. A finished project shall be read only such that the purposes of creating its documentation can be fulfilled.

4.10. Process Improvement Proposals.

It is essential that the PIP be made available at any time, whether or not a project is opened. This is because PIP forms are independent of the current list of projects. A programmer may accumulate many PIP documents in a career.

4.10.1. Use Cases for PIP Forms.

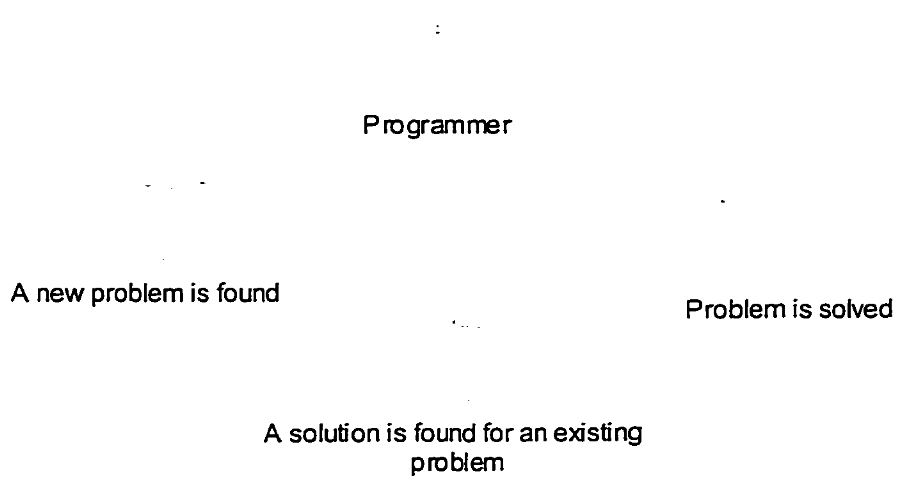


Figure 11. Use case diagram for PIP form management.

The user maintains one list of PIP forms with each instance of a PIP form representing one problem with a possible solution. The PIP forms for a user are managed as per a list, requiring three use cases as documented in Figure 11.

4.10.1.1. Use Case: A new problem is found.

When a process problem is found, the user can request the list of PIP forms and request that a new form to be added to the list. The user edits the PIP to define the new problem. If a solution is known, it is also added. A field is also available for comments. The addition of a PIP document complements a change in the process phase list, checklists, defect types or metrics list.

4.10.1.2. Use Case: A new solution is found for an existing problem.

The user can request the list of PIP forms, select a particular form and request it be edited such that the solution can be documented.

4.10.1.3. Use Case: Problem is solved.

When the problem is solved, the user can request that the applicable PIP form be deleted from the list.

4.10.2. Requirements for PIP forms.

Five requirements were elicited for managing the PIP forms.

Requirement 4.10.2.1. PSP Manager shall provide functionality to allow the user access to his Process Improvement Proposals at any time.

Requirement 4.10.2.2. Each PIP form shall consist of a problem, a possible solution, and a programmer comment.

Requirement 4.10.2.3. PSP Manager shall allow the user to add a PIP at anytime.

Requirement 4.10.2.4. PSP Manager shall allow the user to delete a PIP at anytime.

Requirement 4.10.2.5. PSP Manager shall allow the user to edit a PIP at anytime.

4.11. Programmer Profile.

Each programmer has a name, programming environment (language), and a supervisor. He also has his own programming habits and will maintain his phase names, defect types, metrics plan, checklists. In addition, the programmer has a list of outstanding projects as well as his historical projects. This data combines to create a programmer profile.

When a user starts up PSP Manager for the first time, PSP Manager will automatically create a programmer profile for him, based on the PSP (Humphrey, 1995a). Appendix A documents the requirements for this profile.

4.12.1. Use Cases for the Programmer Profile.

The three use cases shown in Figure 12 are required for accessing the Programmer.

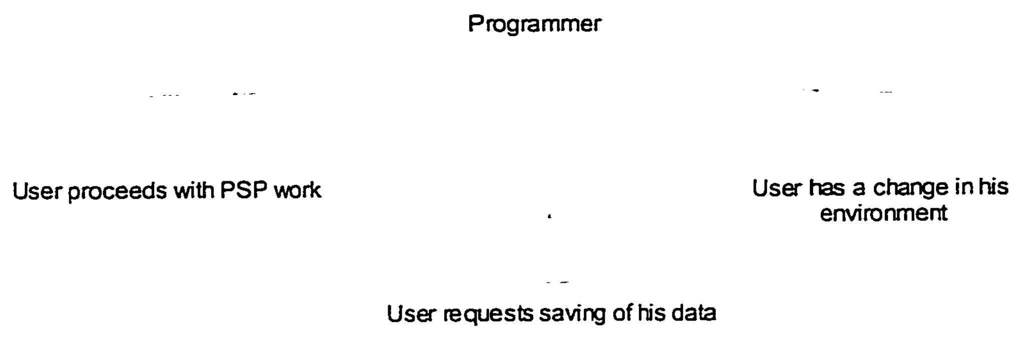

Programmer
User proceeds with PSP work
User has a change in his environment
User requests saving of his data

Figure 12. Use case diagram for the Programmer profile.

4.11.1.1. Use Case: User proceeds with PSP work.

The user starts up the PSP Manager when he is ready to start PSP work. If many users use the same machine, each selects his own PSP directory to identify himself. If a programmer profile is not found, PSP Manager assumes a new user has signed on and will create one. If a profile is found but cannot be loaded, a message is issued back to the user such that the profile can be fixed or restored from backup.

The user is then clear to do work on his projects. When his PSP work is done, he shuts down the tool. When the tool is closed, the current project is closed and the programmer profile is saved to disk.

4.11.1.2. Use Case: User requests saving of his data.

Even though saving can be done automatically throughout run time, the user can request his PSP data be saved. This option minimizes the loss of data in event of an accidental failure.

4.11.1.3. Use Case: User has change in his environment.

The programmer's may request that his or her name, supervisor or language may be changed because of marriage, a change in personnel, or a change in technology respectively.

4.11.2. Requirements for the Programmer Profile.

Seven requirements must be fulfilled to implement the programmer's profile.

Requirement 4.11.2.1. PSP Manager shall maintain a Programmer Profile for the user.

Requirement 4.11.2.2. The Programmer Profile shall include the programmer's current and finished projects.

Requirement 4.11.2.3. The Programmer Profile shall include the programmer's process phases, checklists, and defect types.

Requirement 4.11.2.4. The Programmer Profile shall include the programmer's process improvement proposals.

Requirement 4.11.2.5. The Programmer Profile shall include the programmer's career "to date" time and defect metrics.

Requirement 4.11.2.6. The Programmer Profile shall include the programmer's career to date user metrics.

Requirement 4.11.2.7. The Programmer Profile shall include the programmer's name, programming environment, and supervisor (if applicable).

4.12. Additional functional requirements.

The following requirements are additional requirements that were not elicited from use case analysis, but must be included such that the proper operation of PSP Manager is achieved.

4.12.1. Publication of forms, spreadsheets.

The Software Engineering Institute (SEI, 1996) insists that all personal process data is private unless its owner provides permission to access it. Access to PSP forms is best done through the publication of results. Each of the PSP forms can be published in a user readable format.

Requirement 4.12.1.1. For the plan summary and career report, PSP Manager shall create the reports to be compatible with the Microsoft Excel spreadsheet package.

Requirement 4.12.1.2. The following forms shall be published as user readable text files:

- Time Recording Log
- Defect Recording Log
- Issues
- Process Improvement Proposals
- Test Report Templates
- Checklists.

4.12.2. Design Constraints.

The following requirement is a constraint that shall restrict the design and implementation strategy for PSP Manager. This requirement is such that the objective of platform independence can be fulfilled.

Requirement 4.12.2.1. With the exception of the spreadsheet component, PSP Manager shall operate as a platform independent application.

4.13. Summary.

This chapter has documented the requirements of the PSP Manager, a tool that will provide automated support for the Personal Software Process. Additional requirements were derived such that the user can customize the tool to support his or her specific personal software process.

The requirements were elicited through the use of use case scenarios. Many of the use cases were based on the review and analysis of the PSP support tools documented in Chapters 2 and 3 (Clemens, 1997, Henry, 1997, Wolesley, 1997).

The requirements elicited in this chapter will be the basis for the design and implementation of the PSP Manager in chapters 5 and 6. Each requirement has been uniquely numbered such that they can be traced and accounted for in the evaluation of the completed product.

Chapter 5. Design.

This chapter will describe the design strategy of PSP Manager based on the requirements elicited in Chapter 4.

To help in making the tool transparent to the task at hand, it must operate on the same machine as an aside to the programmer's development environment. Therefore, it has been decided to use Java as the development language such that platform independence (Requirement 4.13.2.1) can be supported. Java applications run through the dynamic loading of classes into a virtual machine (Flanagan, 1997), which is available for multiple computer environments including PC, Macintosh, and UNIX. Java classes and the programmer's profile can be moved between platforms if necessary.

The balance of this chapter is devoted to the design of and the relationship between the PSP Manager classes. The design chapter will use a top down approach. The programmer profile will be shown first. The chapter will also include the design of the PSP Project, the timing strategy, the design of metrics, and the layout of the spreadsheets. The remaining components of the PSP Manager will be described in a more precise detail during the implementation description in the next chapter.

While the actual user interface is not covered in the requirements, the chapter will conclude with a layout of the user interface for the PSP Manager's main window.

5.1. The `Programmer` class.

The programmer profile contains everything pertaining to its user. This includes the user's projects, career performance metrics, and process data. The class `Programmer` will represent the programmer profile (Requirement 4.12.2.1). The Programmer has three attributes: name, supervisor, and programming language (Requirement 4.12.2.7). If a programmer uses many programming languages, it is best to maintain a profile for each language since process data is directly affected by language.

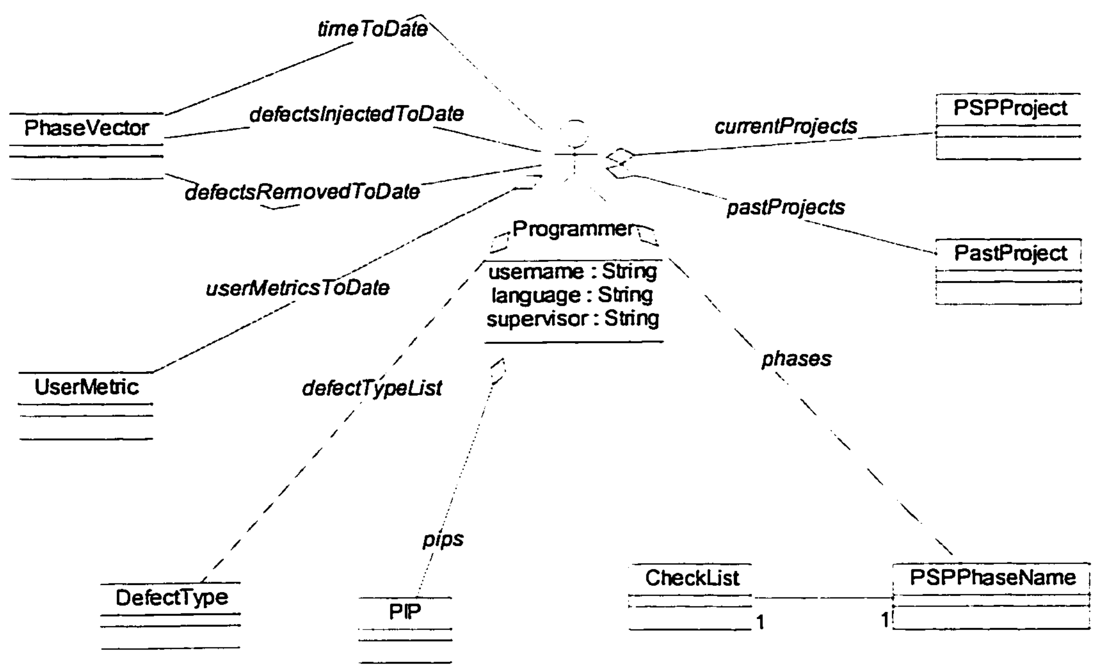


Figure 13. The Programmer Profile.

The programmer can gain access to projects that are outstanding as well as completed (Requirement 4.12.2.2). The class `PSPProject` shall represent a project while a completed project will be represented under the class `PastProject`, which implements queries on the data of a completed project.

The programmer's defect types, phase names and checklists are stored in the programmer profile (Requirement 4.12.2.3). The class `DefectType` stores the defect type information. A phase name in PSP will be stored in the class `PSPPhaseName`. The class `CheckList` represents a list of items against which a completed work product must be reviewed. A Process Improvement Proposal will be stored as an instance of the class `PIP` (Requirement 4.12.2.4).

The Metrics Plan of PSP requires two classes. The class `PhaseVector` represents a list of metrics where each value is tied to an instance of `PSPPhaseName`. Measurements of time, defects injected, and defects removed are sorted by phase. The profile stores three instances of `PhaseVector` to store the "To Date" metrics of time elapsed, defects injected, and defects removed (Requirement 4.12.2.5). This brings the potential of problems when phase names are changed. At the project level, `PhaseVector` instances are automatically updated as project phase lists are updated. When projects are finished, the changed phases show up in the career report undesirably. Phase name changes may also require changes in the formulae used in the spreadsheets.

For metrics not sorted by phase, the class `UserMetric` is used for this purpose. The user can define a name, measurement unit, value and formula for this type of metric. The programmer maintains a list of `UserMetric` instances as the programmer's metrics plan (Requirement 4.12.2.6).

5.2. The `PSPProject` class

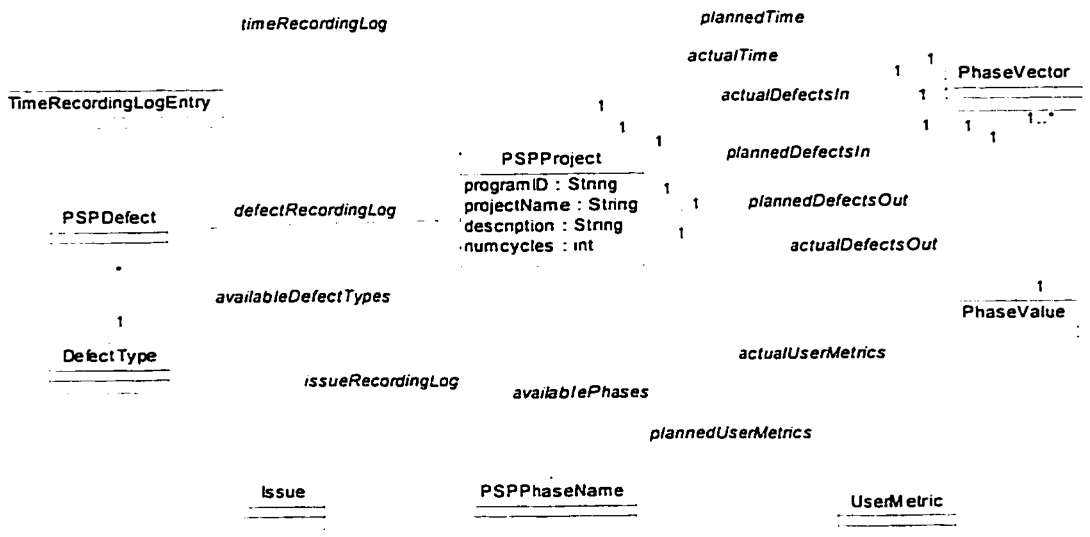


Figure 14. The PSP Project and its relationships.

The structure of the PSP Project is shown as Figure 14. The project consists of many time recording log entries, defects, and issues (Requirements 4.10.2.1, 4.10.2.2, and 4.10.2.3). An entry in the Time Recording Log will be stored in the class

TimeRecordingLogEntry. Each defect shall be stored as an instance of PSPDefect. Each issue shall be stored as an instance of Issue.

The project maintains a pair of instances of each metric in the metric plan (Requirement 4.9.3.4). Time and defect data are stored in the PhaseVector class while the user metrics are managed as a list of instances of UserMetric.

5.3. The Timer class.

The class PSPPhase is a specialization of PSPPhaseName. Its purpose is to support PSP3 through appending a cycle number to the PSP phase name.

The class Timer will be given the responsibility to build the project's time recording log. A Project's Time Recording Log is stored as an instance of TimeRecordingLog.

The Timer does not have access directly to the list of phases, but is provided with an instance of PSPPhaseManager. The timer can query the phase manager for the starting phase, ending phase, the phase after a particular phase, or a phase before a particular phase.

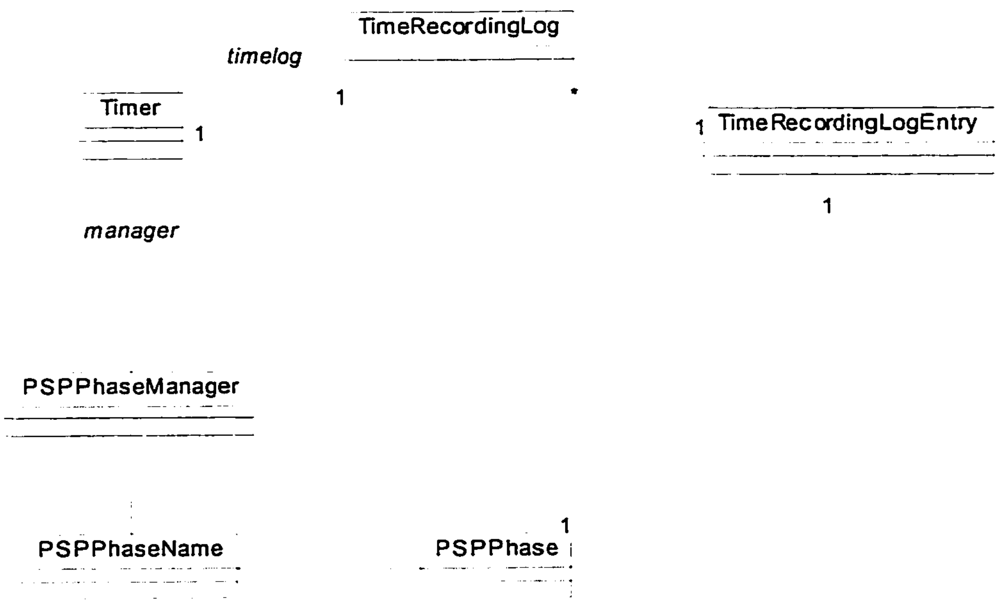


Figure 15. The relationship of times with phases.

The programmer may be working, idle, or interrupted. This leads to timer states of running, stopped, and interrupted respectively. The state diagram for the timer is shown in Figure 15.

Instances of `TimeRecordingLogEntry` can be created in two possible state transitions (Requirement 4.4.2.1). When the state changes from "running" to "stopped", this implies the end of a task and the time must be recorded. An entry is also added when a "change phase" transition happens when the timer is in the "running" state. Here, the timer does an implicit stop and start sequence. If a programmer is disrupted many times in the same task, the timer will accumulate disruption time (Requirement 4.4.2.3).

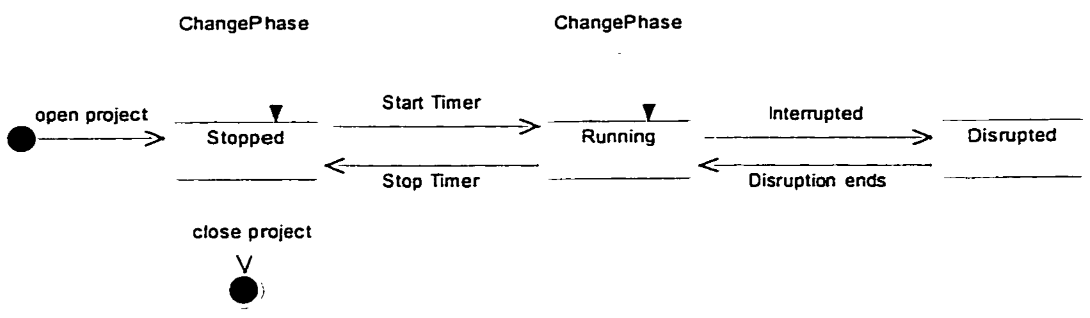


Figure 16. State Diagram of Timer.

The methods that are required for the Timer are shown below.

- `start()`.
- `stop()`: An instance of `TimeRecordingLogEntry` is added to the Time Recording Log when the timer is stopped.
- `disrupt()`.
- `enddisrupt()`.
- `nextPhase()`: The next phase is queried from the phase manager and a call us made to changePhase with this phase as its argument..

- `changePhase(PSPPhase)`: If the timer is running, an instance of `TimeRecordingLogEntry` is created for the current phase before the phase is changed and the timer restarted for the new phase.
- `getCurrentPhase()` returns the current phase that the timer is set at.
- `getState()` tells whether the timer is running, stopped or interrupted.

5.4. Metrics Design.

Figure 16 shows the design for implementing the PSP Metrics Plan. This design is based on the fact that we have two distinct types of metrics that are calculated in the PSP. First, there are phase-related metrics, which are collected from the Time and Defect Recording Logs. Secondly there are user-defined metrics, which may or may not be computed from the phase-related metrics.

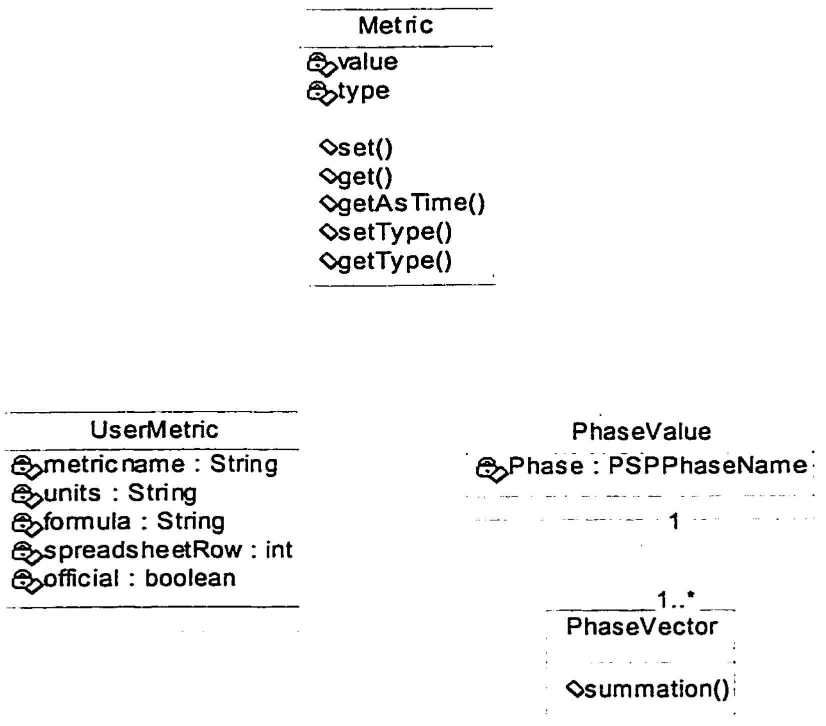


Figure 17. Design of the PSP Metrics Plan.

The base class for the metrics plan will be called `Metric`. Each instance of `Metric` has a numeric value that may be a real number, an integer, or an elapsed time (Requirement 4.6.2.4).

The class `UserMetric` extends the `Metric` class by applying a user defined name and measurement unit (Lines of Code) to a metric value. A `UserMetric` instance may contain a spreadsheet formula instead of a value. The PSP Manager will automatically assign a spreadsheet row to each UserMetric instance.

The class `PhaseValue` connects an instance of `PSPPhaseName` (a process phase) to a metric value. `PhaseValue` instances collect either time or defect counts. Each `PhaseValue` belongs to a `PhaseVector`. The instances of `PhaseVector` are built directly from the list of available phases.

5.5. Design of the spreadsheet.

PSP Manager is required to publish reports that can be converted to a Microsoft Excel spreadsheet (Requirement 4.11.3.1). It was decided to use Microsoft Excel as the default spreadsheet package.

PSP Manager will produce two spreadsheet reports. There is a project plan summary and another for a career report. Appendix B shows the layout of the spreadsheets that will be developed by PSP Manager. The tool will allocate the first eight rows of the spreadsheet for the header of the program. Row 9 will be saved for the total time and defects. The actual metric values will start in row 10. Table 7 displays the usage of the columns for the Project Plan Summary spreadsheet.

The ordering of the phases in the spreadsheet will be the same as in the project's phase name list. The row number for a phase is calculated by taking the position of that phase in the order and adding nine to that number. Therefore the maximum number of rows in the spreadsheet will be the higher of:

- the number of phases plus nine.
- the highest assigned spreadsheet row in the user's metric list.

Column letter.	Column Usage Project Plan Summary	Column Usage Career Report
A	User Metric titles	User Metric titles
B	Planned values of User Metrics	Career values for user metrics
C	Actual values of User Metrics	kept blank
D	kept blank	kept blank
E	PSP Phase names	PSP Phase names
F	Estimated Time in Phase	Total career time spent in each phase.
G	Actual Time in Phase	Percentage of time spent in each phase
H	Estimated Defects Injected by Phase	Total Defects Injected in each phase.
I	Actual Defects Injected by Phase	Percentage of total Defects Injected in each phase
J	Estimated Defects Removed by Phase	Total Defects Removed for each phase.
K	Actual Defects Removed by Phase	Percentage of total Defects Removed in phase

Table 7. Column Layout for Excel Worksheets.

5.6. User Interface layout.

This section shows the main window of PSP Manager followed by a documentation of the menu items. This will be followed by a look at the Test Case window. It follows a baseline that will be used for the interface to the PSP forms.

Figure 17 provides the main PSP Manager window. A button is included for switching the timer on and off. A second button is used to toggle interruption mode. The phase list and available cycles are set up in the two pick lists. The timer can move to the next phase by clicking on "next phase". The timer can be moved to any other phase by selecting the phase and clicking on "go to phase". The remaining buttons provide access to the forms required by PSP.

A problem is noted that clicking on "Go To Phase" must follow a selection of a new phase and cycle in order for the new phase to be timed. Taking an action when the phase is actually changed in the list can alleviate this problem. However, this can cause extraneous entries in the Time Recording Log if the user selects a wrong phase or cycle and corrects it. The delete functionality in the time recording log can be used to remove such entries.

The Help Menu contains one command called "About", which displays a message box describing the PSP Manager.

The History Menu has two commands:

- **Past Projects**: This command brings up the list of past projects for publishing purposes (**Requirement 4.9.3.10**).
- **Career Report**: This command publishes a spreadsheet report summarizing the accumulation of PSP data for all completed projects.

The layout of the three remaining menus is provided in four Tables. Table 8 covers the File menu. Table 9 covers the Project menu. Table 10 covers the Logs sub-menu of the Process menu. Table 11 covers the History menu.

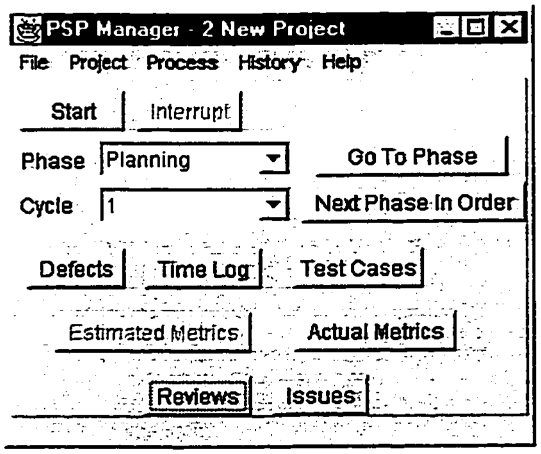


Figure 18. PSP Manager's main window.

Command	Purpose	Requirements Fulfilled
Open Project	Brings up the list of current projects in a window such that a project can be opened. New projects can be created in this window.	
Close Project	Closes the current project.	
Finish Project	Marks the current project as complete. The project is moved to the past project list and the user's career metrics are updated.	4.9.3.9
Save	Saves the current state of the programmer profile	4.11.3.9
Programmer Properties	Allows the user to edit his name, supervisor, and programming environment.	4.11.3.7
Exit.	Quits PSP Manager, stopping the timer, closing the current project, and saving the programmer profile.	

Table 8. File Menu Contents.

Command	Purpose	Shortcuts	Requirements Fulfilled
Phases for the Project	Brings up a window for the phase list for the current project. A phase may be changed at the project level.	None	
Plan Summary	Brings up the Plan Summary window, covering the project name and estimates of phase level metrics.	None	4.9.3.5.
Logs	Serves as a sub menu to access the project related data forms.	None	None
Planned Metrics	Brings up the list of user metrics with the planned values for the project.	"Planned Metrics" button in main window.	4.5.3.2.
Actual Metrics	Brings up the list of user metrics with the actual values for the project.	"Actual Metrics" button in main window.	4.5.3.3

Table 9. Project Menu layout.

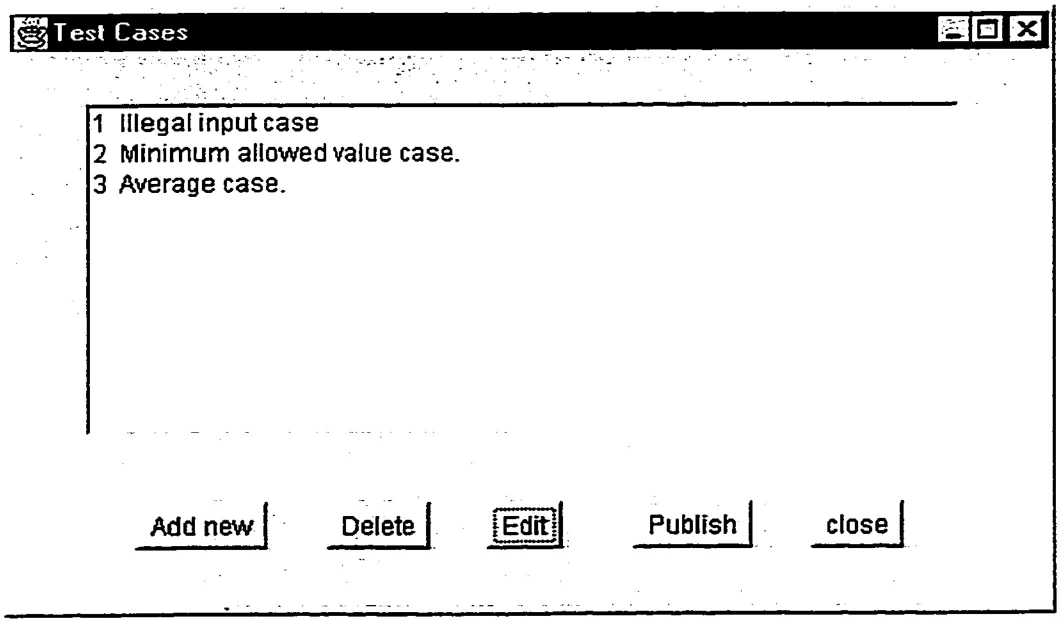
Command	Purpose	Shortcuts	Requirements Fulfilled
Logs: Time	Brings up the project's Time Recording Log.	Control-T or "Time Log" button on main window	4.3.3.14, 4.3.3.16
Logs: Defects	Brings up the project's Defect Recording Log	Control-D or "Defects" button on main window	4.4.5.1, 4.4.5.2.
Logs: Issues	Brings up the project's Issue Recording Log	Control-I or "Issues" button on main window	4.6.3.1, 4.6.3.2.
Logs: Test Cases	Brings up the list of test cases in the project.	"Test Cases" button on main window.	4.8.3.1.

Table 10. Menu commands for the Logs sub-menu of the Process menu.

Command	Purpose	Requirements Fulfilled
Phases	Opens the master list of phases in the process	4.2.3.3.
Defect Types	Opens the list of user's defect types.	4.4.3.1.
Metrics	Opens the master list of user metrics.	4.5.3.1.
PIP	Opens the list of process improvement proposal forms.	4.10.3.1
Check Lists	Opens the user's checklists.	4.7.3.1.

Table 11. Process Menu Layout.

Lists are used to organize PSP data, making the layout of the User interface revolve around the ability to perform operations on lists. Figure 19 shows an example of a list manager.



1 Illegal input case
2 Minimum allowed value case.
3 Average case.

Figure 19. List Manager for Test Cases.

Many requirements specified in the Requirements elicitation specify the addition, deletion, and modification of components of the user's project and process. The List Manager contains five common buttons:

- *Add new* adds a new instance of a PSP class to the list being managed.
- *Delete* deletes the selected instance of a PSP class.
- *Edit* allows the user to edit the contents of the selected instance. A dialog specific to the class of that instance will appear. Changes in the dialog may be accepted or canceled.
- *Publish* allows for this data to be published to a user readable format.
- *Close* closes the window.

Each list manager window will be specialized to manage a list of instances of one PSP class. If a list requires additional operations, additional buttons will be available on the window that manages that type of list.

5.7. Summary.

This chapter provided a top down overview of the design of PSP Manager, which was based on the requirements elicited in Chapter 4. This design shows the relationship between the PSP data items. The chapter included a standard for the layout of the published spreadsheets, the user interface, and the list management techniques available to the user.

While all the components of the PSP Manager appeared at the design level, some of the components are more easily discussed at an implementation level in chapter 6.

Chapter 6. Implementation.

In Chapter 5, the layout of PSP Manager was discussed. The implementation level provides a more detailed description of how the tool will operate.

This chapter provides the key components of the implementation of the PSP Manager classes. Of specific interest will be how the PSP data elements are organized into its various lists. The chapter takes a brief look at each of the lists and the queries required of each list. The chapter will also take a look into the implementation of the class `DeltaTime`, the building of the Plan Summary, the publication of results, and how the programmer profile is written to disk.

The tool and its user interface were developed using Borland J-Builder version 1.0 on an IBM PC clone. The compiled code will be combined into one archive file to support portability between platforms.

6.1. The list classes.

Proper organization of the PSP data will make the PSP Manager code maintainable and allow the application to be efficient. The PSP data consists of many different types of data including log entries, times, test cases, metrics, and issues.

Each PSP data instance will be organized into a list that exclusively handles instances of that class. This is required because each PSP data list has operations that are specific for that type. In Tables 12 and 13, the list classes are shown with the instances that they can handle.

The functions common for each list class are listed below. Additional methods sensitive to a specific list may be added.

- `add(instance).`
- `delete(instance).`
- `size().`

- `at(int)`: This method returns the instance at the given index. The list is indexed from 0 to `size()` less 1.
- `seekForString(String)`: Each of the classes managed by a list has a method `toString()` which returns the string representation of an instance. The method `seekForString` returns the object that produced the given string as the result of calling its toString method. An exception is thrown if no such instance is found.

List Class	Class whose instances are handled by this list	Purpose
`PSPPhaseNameList`	`PSPPhaseName`	This class manages the phases in the Programmer's process.
`TimeRecordingLog`	`TimeRecordingLogEntry`	This class implements the Time Recording Log
`DefectRecordingLog`	`PSPDefect`	This class implements the Defect Recording Log
`DefectTypeList`	`DefectType`	This class manages a list of user defect types
`IssueRecordingLog`	`Issue`	This class implements the Issue Tracking Log
`UserMetricList`	`UserMetric`	This class manages the list of user defined metrics

Table 12. List classes and the instances they handle.

List Class	Class whose instances are handled by this list	Purpose
`PhaseVector`	`PhaseValue`	This class implements a list of metrics designated by a phase name. The values for these metrics are derived from logs.
`PIPList`	`PIP`	This class implements the list of Process Improvement Proposals
`ListOfPastProjects`	`PastProject`	This class manages the user's past projects.
`ListOfProjects`	`PSPProject`	This class manages the user's current projects
`CheckList`	`CheckItem`	This class implements a checklist.
`CheckListManager`	`CheckList`	This class manages all of the user checklists.
`TestCaseList`	`TestCase`	This class manages the Test cases for a project.

Table 13. Additional list classes in PSP Manager.

In C++ these lists could be implemented using templates. The limitations of Java force us to implement each of the lists explicitly. A base list class that implements add, delete, size, at, and seek could have been implemented. Specialization classes for each PSP data storage class could also have been written, however the base class would have to store objects under the Object class name rather than the actual class name. Under that strategy, objects would have to be type cast to their proper class names before use.

6.1.1. `PSPPhaseNameList / PSPPhaseName.`

`PSPPhaseNameList` is the class that manages the process phases. An instance of this list stores the user's process in the profile. It is copied as the phase list when the user creates a project. Specific functions are required for the phase list. They are as follows:

- `startingPhaseName()`
- `endingPhaseName()`
- `nextPhaseName(PSPPhaseName)`
- `previousPhaseName(PSPPhaseName)`
- `clone():` This method allows `PSPPhaseNameList` to create a copy of itself. The result is a deep clone where each phase name in the list is copied.
- `moveUp(PSPPhaseName):` The given phase is moved up one position in the list. This method returns a `void`.
- `moveDown(PSPPhaseName):` The given phase is moved down one position in the list. This method returns a `void`.

6.1.2. `TimeRecordingLog / TimeRecordingLogEntry.`

The `TimeRecordingLog` class manages a list of instances of `TimeRecordingLogEntry`. The contents of the Time Recording Log entry were shown in Chapter 2 (Table 3). The date of the entry is determined by extracting the date from the `start` attribute of the entry as the entry was initialized on this date. PSP Manager assigns a unique identifier to each instance of `TimeRecordingLogEntry` for the purposes of searching.

`TimeRecordingLog` requires additional functionality such that queries can be made on the amount of time spent. The Plan Summary window provides three queries, which are made self-explanatory by their method names.

- `getTimeForPhase(PSPPhaseName)` returns the total time recorded in the log for the given phase
- `getTotalTime()` returns the total time recorded in the log.

- `publish()` converts the time recording log into a report.

6.1.3. `DefectTypeList / DefectType.`

The `DefectTypeList` manages the user's defect categories. Defect Types are discussed in Chapter 2. The `DefectTypeList` must assure that no two instances of `DefectType` in the list have the same identification string.

Through the exception handling capabilities in Java, identifiers for new defect types can be computed. To determine an ID for a sub-type an existing defect type, a whole number is concatenated to the ID of the defect that is to be sub-typed. PSP Manager starts with 1 and increases this number until the defect type is added to the list. Assigning an ID to a new defect type requires the identification of the lowest whole number that is not already in the list.

6.1.4. `DefectRecordingLog / PSPDefect.`

The `DefectRecordingLog`, class manages the defects found in a project. The design of the defect recording log is shown in Chapter 2 (Table 4). PSP Manager automatically assigns unique identifiers to each defect.

Three queries can be placed on the Defect Recording Log in the same way as the Time Recording Log. While the size of the list determines the total defect count, the Plan Summary window will request for a defect report by phase. In addition, the user may request a publishing of the Defect Recording Log.

Defect fix time is accumulated as the Defect is being edited. A `GregorianCalendar` instance is created to log the time of the opening of the editor window with another being created when the defect is saved. A `DeltaTime` is computed from the `GregorianCalendar` instances and added to `fixTime` when the defect instance is updated.

The following three queries will be performed on the Defect Recording Log.

- `getDefectsInjectedForPhase(PSPPhaseName)` returns the total defects injected in the given phase.
- `getDefectsRemovedForPhase(PSPPhaseName)` returns the total defects removed in the given phase.
- `publish()`:converts the Defect Recording Log into a report.

6.1.5. Checklist Management.

A user will have many checklists in his process, and each checklist will consist of many items. This leads to a tree structure that is diagrammed in Figure 20.

PSP defines three classes for checklist management.

- `CheckItem` to represent a single check point.
- `CheckList`: associates many `CheckItem` instances with a `PSPPhaseName`.
- `CheckListManager` manages all of the checklists for the user. Checklists are automatically added as new phases are introduced. A checklist cannot be deleted if there are any projects that involve the phase associated with the checklist.

Figure 20. Checklist relationships in PSP Manager.

6.1.6. `IssueRecordingLog / Issue`.

The class `IssueRecordingLog` manages a list of instances of the class `Issue`. Since the entire IssueRecordingLog is usually published in one report, a `publish()` method has been added to `IssueRecordingLog` class. The date and phase resolution fields will remain empty until the user marks the issue as resolved.

6.1.7. `TestCaseList / TestCase.`

Each instance of `PSPProject` contains an instance of `TestCaseList` that stores the test cases for the project. The test cases are all combined into one report which can be published through the `publish()` method in class `TestCaseList`.

6.1.8. `PIPList / PIP.`

The programmer profile contains an instance of `PIPList` that is used to store the programmer's process improvement proposals. They can be accessed at any time PSP Manager is running. PIP forms are not to be confused with project issues. It is assumed that each problem is associated with one suggested solution. Since PIP form in PSP is published individually the `publish()` method for PIP forms will be in the `PIP` class instead of the `PIPList` class. The user can select `PIP` instances to publish in the `PIPList` manager window and click on the publish button.

6.2. The `DeltaTime` class.

`DeltaTime` will store time spans to a precision of seconds, presenting them in the format HH:MM:SS. Time spans can be found in the Plan Summaries, the Time Recording Log, and the fix time of `PSPDefect` instances. The timer also uses it to accumulate interruption time.

6.3. The creation of a new project.

A new instance of PSPProject is created based on the contents of the process stored in the programmer profile. The phase list for a new project is copied from the programmer profile. This phase list is then the basis for developing instances of PhaseVector to store time and defect data from the data logs. Two copies of the metrics plan are created from the programmer profile for storing planned and actual metrics. To collect data, space is allocated for instances of `TimeRecordingLog`, `DefectRecordingLog`, IssueRecordingLog and TestCaseList. The project will refer to the programmer profile for checklists and defect types.

6.4. Conversion of a PSPProject into a PastProject.

When the user declares a project as finished, it is converted into an instance of PastProject. The PastProject will provide the following queries such that historical analyses can be used towards the estimation of future projects (Humphrey, 1995a)..

- getEstimatedTime() returns the estimated total project time in minutes.
- getActualTime() returns the actual total project time in minutes.
- getEstimatedSize() returns the estimated official size of the project.
- getActualSize() returns the actual official size of the project.
- publish() will provide a complete report on the project.

6.5. Building the Plan Summary.

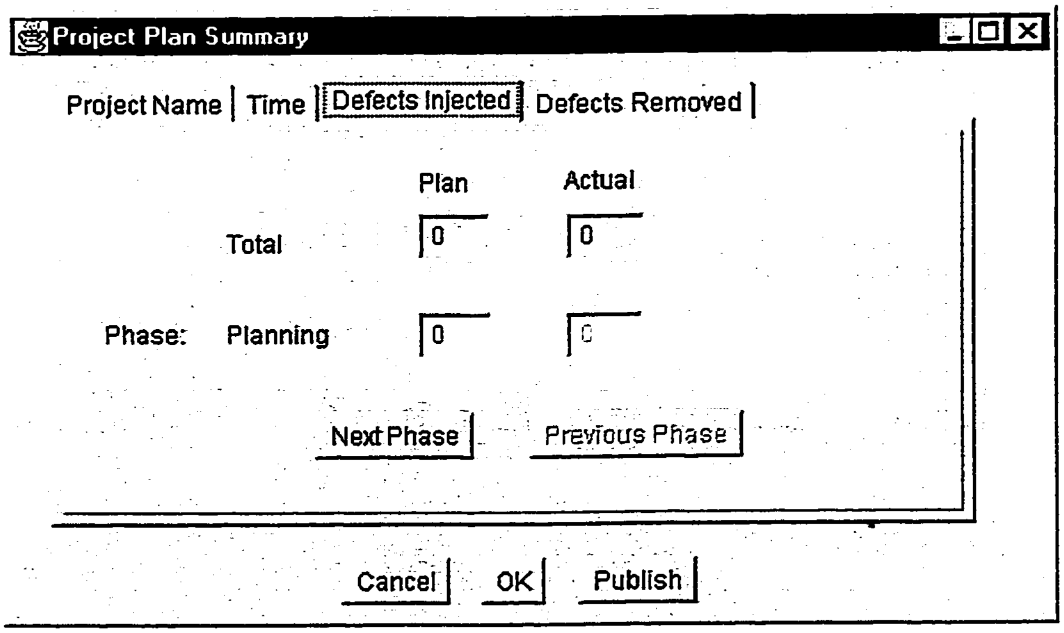


Figure 21. Plan Summary window.

The Plan Summary window (shown as Figure 21) can be summoned at any time by selecting "Plan Summary" from the "Project" menu. The Plan summary has two purposes: to maintain the project's identification data and to allow the user to estimate the time and defects by the phase.

The Plan Summary window consists of four panels. The Project Name panel is for the project name details while the other three panels are used to collect estimates of the time, defects injected, and defects removed respectively. Only the estimate for each phase can be edited. The other fields in the panel are there for comparison. Clicking on "Next Phase" or "Previous Phase" will move the estimation panel through the phases.

The Plan Summary window uses copies of the actual project estimate instances of PhaseVector in the project. This allows changes to be discarded if required. Clicking on Publish will create the Plan Summary as a Microsoft Excel spreadsheet.

6.6. Publishing.

The PSP Manager will provide two types of report. There are the PSP forms such as the Time Recording Log and spreadsheet reports such as the Plan Summary window. All reports will be created as text files on disk. Table 14 provides the default file names for the naming of the report files. A helper application can be opened to read, edit and save these files. Spreadsheet reports will use the comma as the delimiting character. When this type of text file is opened in the package, the user will be queried by the spreadsheet as to how to import the text.

Type of file.	File name format.
Printable text report	Username.projectname.formtype.txt
Microsoft Excel ready text delimited spreadsheet.	Username.projectname.formtype.xls.txt

Table 14. File naming standards.

6.7. Saving the Programmer Profile to disk.

Java provides a persistence model through the `Serializable` interface. Instances of a class that implement `Serializable` can be written to an output stream (Flanagan, 1997). In the case of PSP, all PSP data is persistent. Saving a persistent object to a file is a three-step process.

- An output file is be opened by creating a `FileOutputStream`. Its constructor requires the name of file to be opened for writing.
- The output file is converted to an instance of `ObjectOutputStream`. Passing the `FileOutputStream` as the sole argument to its constructor makes an instance of this class.
- The object is written to the disk by calling `oos.writeObject(target)`, where `oos` is the instance of `ObjectOutputStream` created in step 2.

The `writeObject` method does a recursive traversal of the target object. As `Programmer` stores all the PSP data related to the user, it is the only object that must be written to the disk.

Loading the object from a file on disk takes on a similar three-step process.

- The input file is be opened by creating a `FileInputStream`. Its constructor requires the name of file to be opened for reading.
- The resulting stream is converted to an instance of `ObjectInputStream`.
- The object is read from the disk by calling `ois.readObject()`, where `ois` is the instance of `ObjectInputStream` created in step 2. This returns an instance of `Object` that can be type cast to an instance of the `Programmer` class.

The loading of the file can encounter possible error conditions. The file may not be found, the contents of the file may have been tampered with, or the file opened may not contain a `Serializable` Java object.

This strategy leads to a brittle persistence model, which lacks versioning and upgrade support. A persistent Java object in a file may not be usable when changes are made to the definition of the classes represented in that object. While the object is readable, the Java virtual machine may not be able to type cast the result to fit the changed class. When this happens, a `ClassCastException` is thrown bearing a message that a local class may have been changed.

6.8. Summary.

This chapter has provided a detailed report on the implementation of the PSP Manager automated PSP support tool. The tool was implemented as a Java stand-alone application using the Borland J-Builder development environment.

PSP data was organized in lists. Each list of PSP data has its own unique queries that can be asked. Absolute times are stored as an instance of `GregorianCalendar`. Time spans are stored in instances of `DeltaTime`. The creation of a project was also discussed. The chapter wraps up with how the PSP Manager will write files to disk.

In Chapter 7, the effectiveness of PSP Manager will be evaluated.

Chapter 7. Evaluation of the implementation.

The chapter will evaluate the PSP Manager implementation. There are three key fronts on which the tool will be evaluated.

The first step will be to go back to the requirements in Chapter 4 and see if every one of them has been fulfilled. If one of the requirements has not been fulfilled, it is necessary to discuss why it was not.

A second evaluation process is made available courtesy of the Software Engineering Institute. This department at Carnegie Mellon University in Pittsburgh, Pennsylvania, USA is responsible for the PSP. They have published a guideline for developing automated support for the PSP (SEI, 1996). The PSP Manager will be compared to these guidelines.

The third evaluation is to look at the critical review of the PSP support tools evaluated in chapter 3 and see how this tool has covered shortcomings of the other support tools.

The results of the evaluation will be the basis for the conclusion in the next chapter, where it will be determined if the aim, objectives, and key tasks denoted in Chapter 1 were successfully fulfilled.

7.1. Fulfillment of the requirements.

The first step in the evaluation is to determine if the requirements elicited in Chapter 4 have been fulfilled. The evaluation of the requirements will be aided by referring to each uniquely numbered requirement in Chapter 4.

7.1.1. Phase Management.

It was determined that the PSP Manager required the management of a list of available phases. PSP Manager uses the Java `Vector` class to maintain the ordering of the phases (4.2.2.1). The phase names can be moved within the vector (4.2.2.7 and 4.2.2.8). If an attempt is made to add a duplicate phase name to the list, an exception is thrown (4.2.2.2). If a phase is renamed, the new name is checked against the rest of the list. If a match is found, the change is not accepted. If the match represents the actual instance being changed (this is the case when phase X is renamed to phase X), no change is made.

The user interface that manipulates the list assures the list is in the proper order through refreshing the list on every update (4.2.2.3). When a phase is added to the list (4.2.2.4), it is added to the end of the list. It is up to the user to rename it and move it into position. A phase can be deleted from the list by highlighting it and clicking on "delete" (4.2.2.5). Using the rename (4.2.2.6) button, the phase can be renamed.

Based on the analysis of the requirements, it is concluded that `PSPPhaseNameList`, its methods and its user interface provide complete fulfillment of the requirements for phase management.

7.1.2. Timing requirements.

Time recording log entries are automatically created as the user stops the timer (4.3.2.1). All fields specified in Requirement 4.3.2.2 are supported in the Time Recording Log entry edit facility.

Interruption time is tracked in the timer as an instance of `DeltaTime`. Interruption time is initialized to zero when the timer is started. As a disruption is ended, the time of the disruption is computed and added to the total interruption time for the task (4.3.2.3). The user interfaces allows the user to request the Time Recording Log at any time in the project. (4.3.2.4) Entries can be edited (4.3.2.5) or deleted (4.3.2.6).

It is therefore concluded that the classes `DeltaTime`, `TimeRecordingLog` and `TimeRecordingLogEntry` combined with the design of the user interface provide complete requirements of the timing component of PSP Manager.

7.1.3. Defect Type Management Requirements.

Access to defect types is done through the user interface (4.4.2.1). A delete method in DefectTypeList fulfills Requirement 4.4.2.2. Requirement 4.4.2.3 is fulfilled through an editing facility in the user interface.

The next available whole number that is not used in the list will be set as the identification number of a new defect type (4.4.2.4). When a defect type is assigned a subtype, a sub-type number is appended to the identification number of the defect to be sub-typed. This constitutes the identification number of the new defect (4.4.3.5). Both cases of adding defect types were successfully tested.

This concludes that the `DefectType` and `DefectTypeList` classes along with the user interface fulfill all defect type management requirements.

7.1.4. The Defect Recording Log Requirements.

The Defect Recording Log is accessible throughout the user interface (4.4.4.1). The edit facility supports all required fields in the defect (4.4.4.2 and 4.4.4.5). The Defect Recording Log has both add (4.4.4.3) and delete (4.4.4.4) methods. A defect may either be fixed or saved without fixing through the user interface (4.4.4.6).

All methods of the `PSPDefect` and `DefectRecordingLog` were successfully tested. It is concluded that the classes `PSPDefect` and `DefectRecordingLog` fulfill all requirements for defect management.

7.1.5. User Metric management requirements.

The user interface provides access to the user's metrics plan at the project plan, project actual and career level (4.5.2.1, 4.5.2.2, and 4.5.2.3). While each data item is stored as a double precision floating point number, each real number in a metric can be converted to an integer or `DeltaTime` instance when required (4.5.2.4).

Two edit facilities are provided for a user metric. One facility edits the metric's name, type and measurement unit (4.5.2.5). The second facility controls the value of a metric (4.5.2.6) as well as its spreadsheet formula. The computation of Metrics from the Time and Defect Recording Log data is done through the `PhaseVector` class (4.5.2.7). As they are required, project relevant instances of the `PhaseVector` class are automatically updated.

Formulae and numbers in the metrics plan can be published in a Microsoft Excel readable format. (4.5.2.8). When the spreadsheet is opened, formulae are automatically computed.

The User has an official size metric. One is marked as such in the default programmer profile (4.5.2.9). The current official size metric can not be taken off the list, however it can be changed (4.5.2.10 and 4.5.2.11).

All methods and use cases were successfully tested and it is concluded the classes `PhaseVector`, `PhaseValue`, `UserMetric`, and `UserMetricList` fulfill the requirements for the PSP Metrics Plan.

7.1.6. Issue Recording Requirements.

The log of issues is available through the user interface (4.6.1.1). All fields of the issue are included in the edit facility (4.6.1.2). The user may add (4.6.1.3), delete (4.6.1.4), and edit (4.6.1.5) any issue in the log. The edit facility allows the user to resolve an issue or mark it as outstanding (4.6.1.6).

All use cases and object methods were successfully tested. It is therefore concluded that the requirements for issue management were fulfilled.

7.1.7. Requirements for reviews.

Access to checklists was made available through a menu command (4.7.2.1). A checklist is automatically created whenever a new phase is introduced into either a project or the overall process (4.7.2.2). Check items may be added (4.7.2.3), deleted (4.7.2.4), or edited (4.7.2.5). A publish command is available for checklists (4.7.3.6).

All use cases were successfully tested and it is concluded the requirements for checklist management were fulfilled. However, there is a concern. A `CheckList` instance cannot be deleted at the time a phase is deleted. This is because there are instances of the phase list at both the project and process level. It is essential to assure that the phase is not in the process or in any current project before deleting a checklist. It is more viable to delete a `Checklist` instance automatically to prevent the user error of deleting the wrong checklist.

7.1.8. Test Case Requirements.

The `TestCaseList` instance for the current project is managed through a user interface that is accessible through the Project menu (4.8.2.1), complete with add (4.8.2.3), delete (4.8.2.4), and edit (4.8.2.5) facilities. Each test contains a state variable that shows whether the test is successful, unsuccessful, or incomplete (4.8.2.2).

The classes and their methods were successfully tested and therefore it is concluded that the classes `TestCaseList` and `TestCase` fulfill the requirements for the Test Case management component of PSP Manager.

7.1.9. PSP Project Requirements.

Requirements 4.9.2.1 through 4.9.2.5 are fulfilled through instance variables in the PSPProject. The Plan Summary window can be opened at any time (4.9.2.6). When the plan summary is opened, new PhaseVector instances are created with up to date time and defect data (4.9.2.7, 4.9.2.8). Through the use of copies of the original estimate vectors in the Plan Summary, any changes in the estimates are discarded if the "cancel" button is pressed on the Plan Summary.

Updating the "To Date" vectors in the programmer profile is a part of the "Finish Project" task (4.9.2.9). A publish method in the resulting historical project makes a complete publication of the project data forms (4.9.3.11).

Therefore, it is concluded that the `PSPProject` class fulfills the requirements for data storage of the PSP project.

7.1.10. Process Improvement Proposal (PIP) requirements.

The `PIP` and `PIPList` classes were successfully tested. This component manages a list of instances that contain text information. A user interface and edit facility was created to fulfill all five of these requirements.

7.1.11. The Programmer Profile Requirements.

The user is assigned a programmer profile that was made persistent through implementing `Serializable` (4.11.2.1). Two vectors were maintained for the programmer's projects: one for the current projects and one for the past projects (4.11.2.2). The user has a master personal process consisting of a master list of phases, checklists, and defect types (4.11.2.3). The user has a facility to maintain PIP forms (4.11.2.4). Three `PhaseVector` instances are used to maintain the career time and defect metrics (4.11.2.5). The user can store his master metrics plan as part of his profile. The values and formulae of the metrics are used for to store career to date data (4.11.2.6). Personal information on language, name and supervisor is stored (4.11.2.7).

All tests passed and it is concluded that the `Programmer` class successfully fulfills the requirements for the Programmer Profile.

7.1.12. Miscellaneous Requirements.

Four additional requirements were elicited for two purposes: publishing and platform independence. They are listed as follows:

If a form is publishable, the interface to manage that form will include a publish button (4.12.1.3). The Plan Summary and Career Report will be spreadsheets (4.12.1.1). The remaining reports will use straight text files for creating reports (4.12.1.2).

Requirement 4.12.2.1 is fulfilled within the limitations of the Java programming language. Command line platforms cannot support the graphics required of the PSP Manager. It requires a graphical user interface as a platform. It is difficult to make the spreadsheet component platform independent. There are many spreadsheet packages available. However, not all of these forms may be able to read the comma delimited text based spreadsheet produced by PSP Manager. The formula language for each spreadsheet value is different. The latter problem can be fixed through adjustments in the formulae in the user metric list. It is assumed that Microsoft Excel will be the spreadsheet package used.

This concludes that all four external requirements and constraints were fulfilled.

Since no problems were found, it is concluded that the requirements elicited in Chapter 4 of this thesis were fulfilled by the PSP Manager implementation.

7.2. Meeting the SEI framework.

The Software Engineering Institute in Pittsburgh, Pennsylvania is responsible for the administration and promotion of the PSP. It has published a guideline for the automated support for the PSP (SEI, 1996). This guideline provides recommendations for the requirements elicitation phase of the PSP tool development life cycle.

The analysis of PSP Manager against the SEI specification will be completed through taking key statements from the specification and explaining how the PSP Manager covers these statements.

7.2.1. End-user customization.

The SEI suggests that some end user customization may be built into the software. They specify examples of customization such as the renaming of programming phases or the development of defect types (SEI, 1996).

The PSP Manager provides end user customization in four ways. The phase list may be expanded and phases may be renamed or deleted. Defect types can be added and deleted. Checklists can be customized to the user's preference through additions and deletions. Metrics in the PSP metrics plan can be adjusted for the user's own practice.

The SEI recommends that the user get a full understanding of the PSP and its framework before making changes to it to support a personalized software process. Since PSP Manager starts the user with a default PSP profile, the user can use the PSP Manager for his first PSP project after the course.

7.2.2. Statements on Data privacy.

The SEI specifies that it is important that a person's PSP data is private with the ability to provide public access to completed forms. It is important that the programmer profile remain secure. While the programmer profile is a binary file on disk, it can be copied by anyone who has access. An assumption has been made for the purposes of PSP Manager. It is assumed that the user will use the security features provided by the hardware, network and operating system for protecting his programmer profile.

Through the use of the publish interface, text files and spreadsheets can be developed. These files and spreadsheets can then be copied or moved to in the user's public domain, FTP, or world wide web directory.

7.2.3. Duplication of Data.

The SEI recommends that PSP data is automatically copied among forms, eliminating the need to copy data twice.

The forms in PSP Manager are automatically updated as they are required. However there is one problem that must be addressed. When PSP Manager publishes a spreadsheet, metrics that are computed by a spreadsheet formula filled by copying the formula to the spreadsheet file. However, PSP Manager has no communication with the spreadsheet package. When the spreadsheet is opened for the first time, the results are not automatically copied back to the PSP Manager metrics. It is up to the user to enter them into the project's metric list before declaring the project finished.

There are three strategies that can alleviate this problem. If PSP Manager strictly adopts the original metrics plan of PSP, the formulae can be hard coded into the implementation. However, forcing one metrics plan limits the user's ability to customize a personal process. Some key metrics required by the user may not be in the standard PSP, and some users may find a number of the metrics in the standard PSP as useless. Since customization is a key theme in PSP Manager this strategy is not viable.

The other alternative is to design a specific formula language within PSP Manager. Similarly, a framework can be derived for the Java application to communicate with the Spreadsheet package to retrieve the calculated values of formulated metrics. Both of these options can be treated as an option for future development.

7.2.4. Users can operate in any phase they think is appropriate.

Even though the layout of the PSP implies an ordering of the phases, the SEI suggests the user be given the ability to operate in the phase of his choice. Phase A may naturally happen before phase B. However the system cannot block the user from performing in phase B if there are no time recording log entries for phase A.

The PSP Manager addresses this statement through the timer's interface. The user can choose which phase and cycle he works in at any time from a picklist and choose "Go To Phase".

7.2.5. On-line help.

The SEI recommends adequate on-line help be made available with the PSP support tool.

"Adequate" can be viewed as ambiguous in this context. Each software engineer has his or her own notion of what is adequate. While there is no explicit help file in PSP Manager, the user interface for PSP Manager is made self explanatory through proper wording of the menu files. In addition, each user-defined string in newly created instances of PSP classes is initialized with a help string, showing the user where to place his data. While these concepts hold true for every graphical user interface, many applications still use help files and windows. Since PSP Manager does not have an explicit help facility, it is concluded this requirement is not fulfilled.

7.2.6. Header information.

The SEI recommends that header information be inserted when any form is generated. The header includes the programmer's name, the date, the project number, and the project name. For some forms the programmer's supervisor and programming language are included.

Each instance of a PSP data form template belongs to either a PSP Project or a Programmer. When any PSP data form or template is published to text or spreadsheet, the header information is collected from the project and programmer classes and inserted at the top of the text file.

7.2.7. Availability of the Log forms.

The SEI recommends four forms should be user-accessible at all times during the development process: the Time Recording Log, the Defect Recording Log , the Process Improvement Proposal, and the Issue Recording Log.

PSP Manager conforms to this statement as shown in Table 15.

7.2.8. Phase and Time Recording.

The SEI states that the PSP support system needs a means to record which phase the user is in (the end-user supplies phase information to the system), and to measure the corresponding elapsed times spent in the phase

Form required by SEI	Menu Selection.
Time Recording Log	Project-Logs-Time Recording Log
Defect Recording Log	Project-Logs-Defect Recording Log
Issue Recording Log	Project-Logs-Issue Recording Log
Process Improvement Proposal	Process-Process Improvement Proposals

Table 15. PSP Manager's instant access to Log Data.

This statement is fulfilled through the timer device. The timer automatically fills out the Time Recording Log as the timer is operated. The user through his selection from the phase picklist supplies the phase information. Timing is done through a start-stop-interrupt state machine.

7.2.9. Project Planning.

The SEI supplies a statement regarding estimation of tasks. "To perform a programming task, the PSP requires that historical data are available (to predict productivity, defect injection/removal rates and code size)." (SEI, 1996).

This statement promotes the PROBE method (Humphrey, 1995a) for the estimation of software size. PROBE takes two inputs: an estimate of the size of a new program and the user's historical database. A regression analysis is done on the historical database to yield a projected size estimate for the new.

While the PSP Manager implements the historical database through the `PastProject` and `ListOfPastProjects` classes, PROBE is not explicitly stated as a requirement in Chapter 4. PROBE has been considered as an optional requirement that is to be considered for future work. Another consideration is that not every programmer may be comfortable in using PROBE. Therefore, when PROBE is available in any automated PSP support tool, there should be no obligation to use it.

PROBE is a useful tool when it is used in the default PSP process with at least three past projects (Humphrey, 1995a). However PROBE can become useless when a customized process has been fitted to PSP Manager. PROBE uses the metric called New and Changed Lines of Code as the unit of size measurement. If a programmer is to change the official size metric, an adjustment or mathematical model may be required to make PROBE effective.

PROBE relies specifically on a linear correlation between the estimated program size and actual final program size. Regardless of how a user measures his program size, if he can establish this linear relationship for his metrics, then PROBE would be effective. Without this linear correlation, PROBE will produce an ineffective projected estimate and a large limit on the error of estimation. If linear correlation is not effective, the user may investigate correlation at a quadratic, exponential or higher level.

7.2.10. Enhanced Issues.

The SEI suggests enhancing Issues in the Issue Recording Log through adding a Resolution Phase. The PSP Manager actually added two fields to the Issue class. In addition to the resolution phase, PSP Manager added an additional string to describe how the issue was resolved.

7.2.11. High Level Design.

High level design covers the development of design documents. At a personal level, the layout of design documents is dependent on the standards of the programmer involved.

In chapter 4, an assumption was made that external software development tools are available for the development of system designs. Under this assumption, the programmer would not use the four design templates in PSP (Operational Scenario, Logic

Specification, State Specification, and Functional Specification). A user should not be required to take a design document and change it to fit the PSP templates.

Modern software design tools can cover the PSP design templates. The Unified Modeling Language (UML) provides an excellent example. The State Diagram, Use Case Diagram, and Sequence Diagram in UML provide a graphical representation of a design that cannot be done using the State Specification, Logic Specification, Operational Specification, and Functional Specification templates in PSP.

Even though all developers do not use UML, there are other CASE tools as well as design tools that serve as a suitable replacement for the PSP design forms. For these reasons, the PSP design templates were not considered as requirements for PSP Manager. However, they can be considered as future extensions that a user should not be obligated to use.

The design review is supported through a separate phase and checklist. Check items are supplied for the design review in the default PSP profile, however the user may change them. Test cases can be developed at this time through use of the Test Report Template.

7.2.12. Development.

The SEI provides the following steps for development.

- Module Design
- Module Design Review
- Coding
- Develop Test Cases.
- Review Code.
- Compile
- Test
- Recycle

The Module Design and its review progress similarly to the high-level design phase. Design at a module level is required for large programs but may not be required for small programs (SEI, 1996).

During the development and testing phases, the main interaction with the PSP Manager is the timing steps. Test cases can be added to the test cases previously developed in the high level design through partial completion of the test report templates. If defects are found, the Defect Recording Log can be updated.

A project can be recycled using PSP 3 and the incremental model of software development. A project is recycled if the product produced at the end of the PSP phase ordering requires additional functionality. The number of cycles can be set in the Plan Summary form. The default number of cycles is 1.

For some projects, a cycle may not constitute a complete iteration of the phase list. If this is the case, some phase and cycle combinations will not be covered and will show zero time in the time log.

7.2.13. Post Mortem.

The purpose of the Post Mortem phase is to summarize the program and compute the metrics required based on data in the Time Recording Log, the Defect Recording Log, and the size data. The SEI provides a PSP Spreadsheet similar to the one being published by the PSP Manager.

The user asks the tool to publish the Plan Summary spreadsheet. The summary is published and the spreadsheet package is opened such that the result can be imported. It is up to the user to port these results back to the project. The user declares the project as finished so that the career data is updated. It is assumed that the user keeps the completed spreadsheets for publishing purposes.

7.2.14. Summary of SEI Requirements.

It is concluded that the PSP Manager supports a subset of the requirements as set out in the SEI guidelines for PSP automated support. The SEI requirements that were fulfilled by PSP Manager covered the basic workflow of PSP.

PSP Manager provides a transparent data collection that can be accessed throughout the development process. PSP Manager was found to have a detailed level of user customization. It gives the user the flexibility to perform in any phase he feels comfortable with. The user has full access to every form through either a menu selection or button. Timing is done dynamically with the automatic creation of the Time Recording Log. Additional fields were added to the Issue Recording Log and Test Report Template forms.

The main drawback of PSP Manager is the need to copy metric data from a completed spreadsheet into the appropriate metric values in the Programmer profile.

A number of components were not covered because they may not be used in a customized personal software development process. The design templates of PSP are not included in PSP Manager under the assumption each programmer has his own external design tools. PROBE was not specifically covered in PSP Manager because it was found that PROBE may not be suitable when the user makes a change to the size measurement proxies.

7.3. Comparison to the other PSP Support Tools.

In Chapter 2 of this thesis, four automated support tools that support PSP were introduced. They are Timelog, Timmie, PSP Studio and psptool 0.6. In Chapter 3 of the thesis, their advantages and disadvantages were discussed. This section is dedicated to seeing if PSP Manager successfully covered disadvantages of the tools while maintaining the advantages. Three of the four tools (psptool 0.6, Timelog, and PSP Studio) will be covered. Timmie is not covered since it is not specifically geared to automate the execution of PSP.

7.3.1. Comparison to psptool.

The main problem with psptool (Wolesley, 1997) was that the Time Recording Log and Defect Recording Log was not editable without complicated file manipulation. It is important to make data editable in event a mistake was done. The PSP Manager alleviated this problem through making this data editable. This was completed through the development of editing dialog boxes for both data types. Access to the defect dialog is standard for all defects. While the actual timer details cannot be edited, the Time Recording Log entries can be edited. Access to the time recording log entry dialog box is optional for each entry.

A historical analysis is made easier in PSP Manager through the `PastProject` class. A past project can be published or simply queried for the relationship between the estimates and actual values of total size and total elapsed time. With psptool, you must manipulate files to get at data from the past projects.

PSP Manager supports many outstanding projects at a time, while psptool only supports one outstanding project.

Since PSP Manager stores all projects in one programmer profile under unique identifiers, there is no risk of clobbering old projects by using the same name as a new project. Since psptool stores each project in a separate file, there is a risk of a clobbered file when a new project is named the same as an old project. The psptool does not query the user of this problem using an "are you sure" dialog.

PSP Manager does not force to user to base estimates of time and defects based on previous "To Date" percentages for setting estimates. This is not possible in psptool. Through avoiding use of the "To Date" methods, the user can specifically allocate the time more specifically to tasks.

Finally, PSP Manager provides text reports or spreadsheets for all PSP data forms, compared to no reports from psptool.

7.3.2. Comparison to Timelog.

Timelog (Clemens, 1997) only covers the Time Recording Log component of the PSP process. Full support of PSP goes well beyond just recording time.

Like Timelog, PSP Manager is a Java application that is platform independent. While Timelog supports multiple processes, PSP Manager provides limited support for multiple processes. PSP Manager maintains a master process that is applied to newly created projects. The master process can be changed between new projects.

The main problem with Timelog is that the timing is confusing compared to PSP Manager. Timelog uses two "Now" buttons to fill the start and stop time fields with time data. A third button (add) must be clicked to add the entry to the log. Interruptions are dynamically timed. PSP Manager uses the time component of GregorianCalendar instances to log the start and end time, computing the time span automatically.

PSP Manager does not inspect time strings until it is necessary to do so. If an illegal time string is found, the user is notified such that he can fix it. Restoring the old time in the case of Timelog causes a total loss of work, even if it could have been fixed with a couple of keystrokes.

Like psptool, Timelog does not print reports. The Time Recording Log in PSP Manager is publishable to a text file.

7.3.3. Comparison to PSP Studio.

In comparing PSP Studio (Henry, 1997) to PSP Manager, two issues are raised: the issue of end user customization and the target population.

The PSP Studio seems to be designed for those who are taking the course. It supports all seven levels of PSP from PSP0 to PSP3. PSP Studio also supports all the available PSP forms and implements dynamic timing. It is specifically targeted to people taking the PSP course as well as the instructors.

PSP Manager is designed to use the PSP as a baseline to develop one's own personal software development process. Its target population is engineers who have already taken the PSP course. Since it is assumed that the target users of the system have taken the PSP course, they will take one version of the PSP process and make modifications to it.

PSP Studio requires a Microsoft Windows platform. It is not platform independent. Even if you use Windows, the organization of the forms is difficult to navigate. PSP Manager makes navigation of PSP data forms simpler through a main window that provides simple access to every PSP data item. PSP Manager includes shortcut buttons to the most important PSP forms. The PSP Studio has a toolbar, but it requires buttons for the key PSP tasks such as toggling the timer on and off, changing phases, and creating new defects.

The main reason that more established engineers would prefer to use PSP Manager over PSP Studio is the fact that PSP Manager allows the user to customize the tool to his or her own development process. The PSP Studio requires the user to follow the PSP to the letter. It supports the customization of defect types only. This is not viable for established engineers, who already have their own metrics plan, phases, and checklists. Also, they may not use some of the PSP features like PROBE and the design templates. They want a support tool that can be fitted to their practice. The PSP Manager provides the users with this functionality. It allows the user to implement process changes and improvements in the process as opposed to proposing solutions. Check lists are available for every phase because some users have processes that use checklists consistently.

7.4. Problems: Data quality.

One of the major concerns in collecting PSP data is the assuring the quality of the data. Disney and Johnson performed an investigation into PSP data quality and found a number of problems that could be encountered (Disney and Johnson, 1998).

The investigation covered hand collection and calculation of PSP data. While many of the problems specified by Disney and Johnson can be alleviated by automated support, there are still some problems that could arise in the data collected by PSP Manager.

- Errors in Time Recording Log Entries.
- The cumulative fix time for defects removed in a phase is greater than the total time spent in that phase.

Since the time of each state change in the timer must be recorded dynamically, the start time is always before the stop time in each entry. The worst case here is that the stop time of an entry can be the start time of another entry (when the phase is changed while the timer is running). Similarly, interruptions can only happen between the start and stop times of the entry. This makes it impossible for an interruption time to be larger than the delta time.

Therefore, the problem of overlapping Time Recording Log entries is impossible in PSP Manager until the user starts editing the entries. The user can inject a defect in an entry in two ways. First, he can set the start or stop time to be between two other entries. A more serious problem evolves when the interruption time is be edited to be larger than the time difference between the two times. The result is a negative delta time, which implies time saved in a phase! While both cases are physically impossible, the latter problem must be prevented because it has a severe effect on the Time in Phase component of the Plan Summary. Checking for overlaps after an edit is possible, but expensive for a large project with a large Time Recording Log.

It is possible in PSP Manager that the defect fix time for all defects removed in a given phase may exceed the total time spent in that phase. While there is no specific analysis of defect fix time in PSP Manager, it is important to be aware of the cause of this problem, which is the opening of defects when the timer is not running. Defect fix times are not tied to the timer but to the life span of the defect windows. This could cause a problem in three ways. First, the user could be interrupted when fixing a defect. Secondly, the user could forget a defect is open. Third, if a defect is saved as unfixed, time to fix that defect is not accumulated until that defect is opened again, even if a fix is suddenly discovered for that defect. This problem may be fixed in two ways. First, the user can be asked enter the fix time explicitly. However the user's estimate of a defect fix time is subjective. The other alternative is to have the timer dispatch events to the defect editing windows as its state changes.

7.5. Summary.

This chapter provided an evaluation of the implementation of PSP Manager. The evaluation was performed on three fronts. First, the tool was evaluated against the requirements elicited in Chapter 4. Second, it was compared with the SEI guideline for PSP automated support. Third, it was compared to three PSP support tools currently available.

It was found that all the requirements elicited in Chapter 4 were fulfilled. Some problems were identified when PSP Manager was compared to the SEI guidelines for PSP automation. Problems were found in regards to the usage of PROBE, the usage of design templates, the lack of on line help, and the transferability of results from a spreadsheet to the PSP Manager classes. These problems are related to the customization component of PSP Manager and could have been alleviated through having PSP Manager make the user follow the default PSP. However implementing the tool to follow the default PSP would violate the customizability of the PSP as requested in section 7.2.1.

It was found that PSP Manager provided coverage for the disadvantages of psptool, PSP Studio, and Timelog, although it was found that some of the advantages were compromised to alleviate the disadvantages.

The chapter wrapped up with a comment on data quality and two problems that will require functional extensions to PSP Manager.

Chapter 8. Conclusion and Future Directions of Work.

This thesis was focused on the development of software tool support for a customized personal software development process. This chapter wraps up the thesis by determining how PSP Manager fulfilled the key tasks, objectives, and primary aim identified in Chapter 1.

A top down approach was used to identify the aim and research objectives of the thesis in Chapter 1. The objectives were further decomposed into key tasks that were required. The conclusion will proceed from the bottom up and evaluate how the tasks were completed.

8.1. Fulfillment of the key tasks.

In Chapter 1, four key tasks were identified from the objectives.

- **Key Task 1**: The problems and challenges of the PSP must be identified such that the motivation to implement automated support for PSP is recognized.
- **Key Task 2**: Strengths and weaknesses of existing PSP support tools must be identified.
- **Key Task 3**: The newly developed tool should overcome shortcomings of the existing PSP support tools.
- **Key Task 4**: The effectiveness of using PSP Manager to execute the PSP process must be determined.

The first key task was to determine why automated support for PSP was needed. This was identified in the literature review during Chapter 2. In addition, the SEI provided motivation to build automated support for the PSP (SEI, 1996). They were already identified in Chapter 1.

The second key task was covered through chapter 3, which included an analysis of four tools that could be used to support PSP. Three of the tools explicitly implemented one or more PSP forms while a fourth could be adjusted to work with the timing of PSP. All four tools were found to have strengths and weaknesses.

The third key task was covered in Chapter 4. The critical review of the PSP support tools provided the foundation for the development of PSP Manager. The strengths and weaknesses of the tools identified in Chapter 3 were determined to identify requirements for PSP Manager. The requirements served as the basis on which the PSP Manager was designed and implemented.

PSP Manager overcame the shortcomings of psptool in several ways. First, its edit facility alleviates the high cost of making a mistake with psptool. The project management component of PSP Manager uses ID numbers to assure that a project is deleted only through the authority of the user. It also supports the user entry of time and defect estimates that psptool does not support.

PSP Manager supports more PSP activities than psptool, Timelog, and Timmie. Unlike Timmie, the user does require an external script or tool to manipulate the PSP Manager data.

Unlike PSP Studio, PSP Manager automatically provides the user with a checklist consisting of items. The user may alter the check items to conform to his practice. PSP Manager also provides shortcut buttons for the key tasks of PSP that are lacking in PSP Manager.

The fourth key task was challenging. Because of time constraints, a user study was not conducted. As compared to executing the PSP on paper, use of the PSP Manager for executing the default PSP hopes to reduce overhead cost in the following ways.

- A form in PSP Manager can be brought up on demand. This is quicker than sorting through PSP paperwork.
- The time tracking was done through an automatic timer, eliminating the need for a timepiece and time calculations.
- Either PSP Manager or the spreadsheet automatically does the calculations. Time and defect data are automatically accumulated through queries on the

logs. This saves a lot of time in the post mortem phase. PSP Manager eliminates the need to count and tally time and defect entries. It also eliminates the need to hand calculate the many metrics required by PSP.

- If there is an error found in the data, the PSP Manager lets the user make the correction. All corrections are automatically projected through the forms. A correction under hand execution of PSP would be time consuming in comparison as metrics and forms would have to be edited.
- The PSP Manager reduces redundancy by making data available to all forms. This prevents the need to copy data by hand.
- The PSP Manager's Test Case facility makes test planning easier through managing the Test Report templates. Having the entire list of test cases handy in the facility makes the user assured that all test cases are completed before the project is finished. Test plans may be done by hand using many Test Report templates, which can be difficult to sort through.
- The tool was found to mitigate inconvenience to the user because it could fit on the computer's graphical desktop with the development environments of the user. The user can easily switch between the environment and the PSP Manager. Performing PSP on paper forces the user to "go out of the way" to record data.

The customization facilities of PSP Manager allow the user to adjust the PSP defect types, phases, checklists, and metrics to their own practice, showing their own problems and their own improvement suggestions. After starting with the default PSP, it is natural that a user will wish to change the process specification to make it more personalized. Adjusting the PSP on paper is difficult because changes have to be reflected in new forms. In the best case, forms are saved on a word processor and printed as needed.

8.2. Fulfillment of the objectives.

Chapter 1 identified three objectives for PSP Manager.

- **Objective 1.** To provide a software tool that provides automated support for the Personal Software Process.
- **Objective 2.** To provide mechanisms that allow a user to adapt the tool to his personal software process.
- **Objective 3.** To make the components of this software tool (except the spreadsheet component) platform independent.

Based on the analysis of the key tasks, it is easy to see how the objectives are fulfilled.

Objective 1 was fulfilled with the following limitations. PROBE was identified as a future non-obligatory component as some engineers are not comfortable with the PROBE estimation method. Similarly, the PSP design templates were declared as a future non-obligatory component based on the assumption that external software design tools were available for the user.

The default programmer profile supplies the phase names, check items, defect types, and metrics that confirm to the specifications of the Personal Software Process (Humphrey, 1995a). This conforms to Objective 1.

Objective 2 is fulfilled through providing the user functionality to change his or her PSP phases, defect types, checklists, and user metrics. Changes in defect types and checklists show in all the projects immediately while changes in the master phase list and user metrics show in the next new project. Phases and metrics can be changed at a project level.

Objective 3 is fulfilled by using Java as the development language and archiving the complete list of class byte code files into one file that can be added to a system's CLASSPATH environment variable. The spreadsheet is designed using Microsoft Excel formulae, but can be adjusted to another spreadsheet language by adjusting the formulae accordingly.

8.3. Fulfillment of the aim.

The aim was identified in Chapter 1 as follows.

- **Aim**: To provide a software tool that provides automated support for the execution of a customized personal software development process using the Personal Software Process as a baseline in developing such a custom process.

It can be concluded that this aim was successfully fulfilled. First, PSP Manager runs in the same environment as the development environment. The user simply has to switch to the PSP Manager to record a time entry, interruption, or defect. Secondly, the tool uses the Personal Software Process as a baseline in developing a user's personal software development process. When PSP Manager starts up, a persistent programmer profile is created based on the specification for the Personal Software Process (Humphrey, 1995a). The tool uses the notion of phases and defect types in the same way as the PSP. The user may change the PSP specification by adjusting the phases to fit his process. In addition, the individual user can adjust the lists of defect types and check items that correspond to common problems. The user can also add his or her specific metrics to the PSP Metrics plan. Similarly, these items can be re-worded or deleted from the process. This indicates the tool supports a customized personal software development process.

8.4. Future directions of Work.

8.4.1. PSP Research.

Based on the experience encountered in this thesis, it will be important to scale the PSP beyond the actual production of software products. The following ideas provide examples as to how the PSP can be enhanced in this way.

- The requirements stage is key part of any software project. It is probably the most important component of the project (MacAulay, 1996). However, the negotiation of requirements is not a phase in the PSP. A problem with including the requirements as a phase in PSP is the uncertainty surrounding the time line of this stage. It is difficult to estimate this time. It would make sense to identify a relationship between time in the requirements phase and the measurements of the actual product including size and development time.
- Software documentation should be counted as a PSP phase as the development of user manuals and on line help are key tasks in software development.
- The PSP Plan Summary Form contains an extra phase in the "Defects Removed" table called "After Development." (Humphrey, 1995a) "After Development" is not considered a phase. It can be upgraded to a post development phase to accommodate maintenance. It is difficult to estimate the time for maintenance.
- For team projects, the PSP can be used as a baseline to develop a set of generally accepted software standards and metrics that is to be used by the entire team. Humphrey is currently working on the "Team Software Process (TSP)," which is expected to apply PSP methods to the team environment. The TSP also addresses the CMM key practice areas of training, requirements management, quality assurance, and inter group coordination.
- The PROBE method of estimation needs to be extended beyond the Lines of Code measurement unit. Relationships between size estimates and actual size measurements of different size units must be identified through the use of mathematical models. PROBE looks for a linear regression between estimated and actual size.

8.4.2. Introduction into Industry

Engineers admit they would take the PSP course for their own benefit. One software engineer confessed "This is not for the company, it's for me" (Humphrey 1995c). The personal benefits realized by individual engineers will be propagated to the company through higher quality work products and an increased level of confidence in the newly trained engineer.

A company may provide PSP training to the entire team. However, they may view the introduction of the PSP methods as a risk. It takes a strong commitment from management and employees alike. It also requires a significant amount of resources (Humphrey, 1995a). With such a cost, the company's instinct is the expectation of a major benefit. Even though PSP is being used in industry, there are no publications on the effects of its application in real life project. Companies will be reluctant to introduce the PSP methods to their employees until such a report is released.

The Software Engineering Institute has developed the PSP Strategic Initiative (Software Engineering Institute, 1996b) which provides a plan to diffuse the PSP practices through the software industry. It is still up to each individual company to decide whether to accept this challenge.

8.4.3. The future of PSP Automated Support.

The development of the PSP Manager provides these opportunities for future research.

One research idea is to determine what environments would be best served by automated support tools. With the demand for automated support originating in the academic community, automated support tools like PSP Manager should be tested in the academic environment as soon as possible. It is important to note differences between performing the PSP using such support as opposed to performing the PSP with pen and paper.

Current research into PSP based methodologies include the One-Person Project and the Team Software Process (TSP). As these processes get put into use, demand will exist to develop automated support for both of them. Since the PSP is the basis for both the One-Person Project and TSP, principles used for automating in PSP can be applied to automated support for both processes. For the One-Person Process, the key point will be whether or not existing PSP support tools such as PSP Manager can be used "as is" or extended to support it. For the TSP, a distributed (preferably web based) system will be required to automate it, as it is a team-based process. The key point becomes how the individual members can be combined into the team data. It is possible that components of PSP Manager can be reused in a web-based automaton of TSP.

8.5. Summary.

This chapter summarizes the research that was performed in this thesis. The conclusion shows how the key tasks, objectives and aim of the research were met. It was found that the implementation of PSP Manager fulfilled the aim of providing a tool to provide automated support for a customized personal process using PSP as the baseline for development.

If someone has previously performed PSP by hand in the past, the PSP Manager tool will provide an appreciation for the overhead cost of performing the PSP. Once a person has a feel for the default PSP, he can use PSP Manager to develop a personal software development process using the phase-based structure of the PSP. He can develop his own phases and metrics as well as defect types and checklists and have project summaries published in the same format as the PSP forms.

In the end, it is up to the individual software engineer to use a specific software development process or a specific tool. Many engineers may be drawn away from PSP with its high overhead cost. Many users receive a bad impression of the PSP because of its paperwork. With the availability of automated support tools like PSP Manager that can cut this overhead cost, it is hoped that more engineers outside the academic environment will be eager to learn and use the Personal Software Process.

References

Bagert, .D. J. (1997) A Comprehensive Integrated Three-Semester Software Engineering Sequence. **Frontiers in Education (FIE), 1997 21st Annual Conference on**. Los Alamitos, California: The Institute of Electrical and Electronics Engineers. http://fairway.ecn.purdue.edu/~fie/fie97/papers/1148.pdf

Boehm, B. W. **Software Engineering Economics.** Englewood Cliffs, NJ: Prentice Hall.

Clemens, C. (1997). **Timelog**. Sankt Augustin, Germany: Multimedia Applications in Tele-co-operation, German National Research Center for Information Technology. http://mats.gmd.de:8080/clemens/java/timelog/index.html

Flanagan, D. (1997). **Java in a Nutshell.** Cambridge: O'Reilly & Associates.

Frankovich, J. F. (1997). **The One Person Project Software Process**. Masters thesis. Calgary, AB: University of Calgary. http://www.cpsc.ucalgary.ca/~johnf/SENG/thesis/thesis.htm

Hayes, W. and Over, J.W. (1997). **The Personal Software Process: An Empirical Study on the Impact of PSP on Individual Engineers.** Technical Report CMU/SEI-97-TR-001. Software Engineering Institute. Pittsburgh, PA: Carnegie Mellon University.

Henry, J. (ed.). (1997) **Personal Software Process Studio.** Johnson City, Tennessee. Department of Computer Science, East Tennessee State University. http://www-cs.etsu-tn.edu/softeng/psp/index.htm

Hilburn, T. and Towhidnejad, M (1997). Integrating the Personal Software Process across the Undergraduate Curriculum. **Frontiers in Education (FIE), 1997 21st Annual Conference on.** Los Alamitos, California. The Institute of Electrical and Electronics Engineers.

Humphrey, W. S. (1989). **Managing the Software Process.** Reading, MA: Addison-Wesley.

Humphrey, W. S. (1995a). **A Discipline for Software Engineering.** Reading, MA: Addison-Wesley.

Humphrey, W. S. (1995b). The Personal Process in Software Engineering. **Proceedings of the Third International Conference on the Software Process, Reston, VA, October 10-11, 1994**, pp. 69-77.

Humphrey, W. S. (1995c) The Personal Software Process. Overview, Practice, and Results. **Software Process Improvement Forum.** Jan.-Feb. 1995, pp. 8-10. Pittsburgh, PA: Research Access Incorporated.

Humphrey, W. S. (1996a). Using a Defined and Measured Personal Software Process. **IEEE Software** 13(5), 77-88

Humphrey, W. S. (1996b). A Personal Commitment to Software Quality. **American Programmer** 7(12), 2-12.

Humphrey, W. S. (1997). **Introduction to the Personal Software Process.** Reading, MA. Addison-Wesley-Longman.

Humphrey, W. S. (1998). **Why Don't They Practice What We Preach?** SEI Article. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/publications/articles/sources/practice.preach/index.html

Institute for Program Structures and Data Organization (1997). **Timmie.** Karlsruhe, Germany. University of Karlsruhe. http://wwwipd.ira.uka.de/%7Egramberg/PSP/#timmie

Leabo, D. A. (1968). **Basic Statistics.** Homewood, IL: Richard D Irwin Inc.

Lederer, A. L. and Prasad, J (1998). A Causal Model for Software Cost Estimating Error. **IEEE Transactions in Software Engineering,** 24(2), 137-147.

MacAulay, L. A. (1996) **Requirements Engineering.** New York. Springer - Verlag.

Paulk, M., Curtis B. Chrissis, M. and Weber, C. (1993). **Capability Maturity Model for Software Version 1.1.** Technical Report CMU/SEI-93-TR-24. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Scacchi, W. (1987). **Models of Software Evolution: Life Cycle and Process.** Technical Report SEI-CM-10-1.0. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Smith, M. R. (1997). Parachuting Software Engineering Practices into the Hostile Environment of a 4th Year Final Term. **Frontiers in Education (FIE), 1997 21st Annual Conference on.** Los Alamitos, California. The Institute of Electrical and Electronics Engineers. http://fairway.ecn.purdue.edu/~fie/fie97/papers/1364.pdf

Software Engineering Institute (1996a) **A Specification for Automated Support for the PSP**. Pittsburgh, PA: Carnegie Mellon University. http://www.sei.cmu.edu/pspAuto

Software Engineering Institute (1998). Why Use the PSP - Overview. Pittsburgh, PA: Carnegie Mellon University. http://www.sei.cmu.edu/activities/psp/WhyPSP.htm

Succi, G. C. and Benedicenti, L. A Framework to Analyze the Impact of Code Reuse on Productivity. Calgary, Alberta: The University of Calgary.

Wolesley, A. (1997). psptool 0.6. Doncaster, Australia. TQC Software Pty. Ltd. 1997. http://www.neosoft.com/tcl/ftparchive/sorted/math/psptool0.5p1/

Williams, L. A. (1997). Adjusting the Instruction of the Personal Software Process to Improve Student Participation. **Frontiers in Education (FIE), 1997 21st Annual Conference on**. Los Alamitos, California: The Institute of Electrical and Electronics Engineers. http://fairway.ecn.purdue.edu/~fie/fie97/papers/1305.pdf

Appendix A. Contents of the Default Programmer Profile.

The purpose of this Appendix is to document the contents of the Default Programmer Profile to be implemented by PSP Manager. This profile provides the Programmer with the original phases, defect types, check items, and metrics provided by the Personal Software Process in the book "A Discipline for Software Engineering" (Humphrey, 1995a). It is allocated to a new instance of the `Programmer` class when it is first created. The Programmer may modify it to meet the needs of his personal software development process.

The checklists are developed on the assumption that C or C++ will be used as the programming language. The programmer may edit these items to adapt the checklists to the language or development environment they are using.

A.1. Phases.

The Default Programmer Profile shall contain the following PSP phases:

- Planning
- Design
- Design Review
- Code
- Code Review
- Compile
- Test
- Post-Mortem.

A.2. Phase Metrics.

The Default Profile shall maintain the following metrics for each of these phases in order to accumulate the Programmer's career totals.

- Total Time spent in phase To Date.
- Total Defects injected in phase To Date.
- Total Defects removed in phase To Date.

A.3. Programmer Defined Metrics.

The Default Profile shall include the user-defined metrics in Tables 16 and 17.

Metric Name	Measurement Unit	Source
New and Changed Size (Official Size Metric)	LOC	Computed by spreadsheet.
Base Program Size	LOC	User entered.
Deleted Components	LOC	User entered.
Added Components	LOC	Computed by spreadsheet
Modified Components	LOC	User Entered
Reused Components	LOC	User entered.
Total program size	LOC	Measured and entered by user.
Total Defect Density	Defects per KLOC	Computed by spreadsheet.
Test Defect Density	Defects per KLOC	Computed by spreadsheet.
Productivity	LOC Per Hour.	Computed by spreadsheet.
Yield (ratio of test defects to total defects)	Ratio.	Computed by spreadsheet.
Cost Performance Index (ratio of planned time to actual time).	Ratio	Computed by spreadsheet.

Table 16. Metrics Plan for PSP.

Metric Name	Measurement Unit	Source
Cost of Quality - Appraisal (percentage of total time spent in reviews).	Ratio	Computed by spreadsheet.
Cost of Quality - Failure (percentage of total time spent in compiling and testing).	Ratio	Computed by spreadsheet.
Appraisal Failure Ratio	Ratio	Computed by spreadsheet.
Defects removed per hour in design review.	Defects per hour	Computed by spreadsheet.
Defects removed per hour in code review.	Defects per hour	Computed by spreadsheet.
Defects removed per hour in compile.	Defects per hour	Computed by spreadsheet.
Defects removed per hour in test.	Defects per hour	Computed by spreadsheet.
Defect Removal Leverage - Design Review to Test	Ratio	Computed by spreadsheet.
Defect Removal Leverage - Code Review to Test	Ratio	Computed by spreadsheet.
Defect Removal Leverage - Compile to Test	Ratio	Computed by spreadsheet.

Table 17. The PSP Metrics Plan (continued)

A.4 Defect Types.

The Default Programmer Profile shall include the following Defect Types, stored under the given identification numbers.

1. Documentation
2. Syntax
3. Package
4. Assignment
5. Interface
6. Checking
7. Data
8. Function
9. System
10. Environment

A.5 Design Review Checklist.

The Default Programmer Profile shall store the following check items as the Programmer's checklist for the Design Review phase.

- All required inputs are furnished.
- All specified outputs are produced.
- All required includes are stated.
- All stacks, queues, lists operate in the proper sequence.
- Recursion unwinds properly.
- All loops are initialized, incremented and terminated properly.
- Proper operation is ensured for the entire range of possible values of each variable.
- Out of bounds and overflow conditions are guarded against.
- "Impossible conditions" are ensured as absolutely impossible.
- Incorrect input is handled correctly.
- Functions, procedures, and objects are fully understood and properly used.
- Externally referenced abstractions are precisely defined.
- Special names and types are clear and specifically defined.

- The scopes of variables and parameters are self evident or defined.
- All named objects are used within their declared scopes.
- The design conforms to all applicable design standards.

A.6 Code Review Check List.

The Default Profile shall store the following check items as the check list for the Code Review phase.

- The code implements the entire design document.
- Any and all includes are complete.
- Parameters and variables are properly initialized when the program starts up.
- All loops are initialized properly.
- All function parameters are initialized properly.
- Function call formats are verified.
- Spelling and name usage is checked.
- All strings are terminated by the null character.
- Pointers are initialized as null.
- Pointers are deleted only after a memory allocation is made to it.
- Memory allocated to new pointers is freed after use.
- The output conforms to the required format.
- Begin and end statements are properly matched.
- The use of logic operators is verified.
- Ensure all parentheses and square brackets are matched.
- Each line is checked to conform to syntax.
- Each line is checked for proper punctuation.
- The code conforms to all applicable coding standards.
- Files are properly declared, opened and closed.

Appendix B. Design of the Spreadsheet Reports.

The purpose of this Appendix is to provide a sample of the two spreadsheet reports developed by PSP Manager. Two types of reports will be generated.

The Career Report (Table 18) publishes the programmer's overall history for all projects that have been completed. The time and defect data are accumulated over the historical projects. A metric called the "To Date Percentage" is used to determine the proportion of the programmer's total time that is spent in each phase. In the Career Report, metrics with user supplied values are accumulated over projects while the formulae are automatically computed by the spreadsheet.

The Project Plan Summary (Table 19) publishes the PSP data for one specific project. Each metric is assigned a planned and actual value,

PSP Career Report									
Name:	Dale Couprie		Instructor:	Maurer					
			Language:	Java					
			Date:	8/17/98					
			Phase	Time In Phase		Defects Injected		Defects Removed	
User Metric Name	Career Value			To Date	To Date %	To Date	To Date %	To Date	To Date %
			TOTAL	3162	100	135	100	135	100
Base Program Size	770		Planning	231	7.305503	3	2.222222	0	0
Deleted LOC	46		Design	418	13.21948	24	17.77778	0	0
Added LOC	871		Design Review	255	8.064516	0	0	17	12.59259
Reused LOC	610		Code	751	23.75079	108	80	1	0.740741
Modified LOC	146		Code Review	331	10.46806	0	0	90	66.66667
New and Changed Lines of Code	1017		Compile	181	5.724225	0	0	6	4.444444
Total LOC	2205		Test	440	13.91524	0	0	21	15.55556
Total Defects Per unit size.	132.7434		Post-Mortem	555	17.55218	0	0	0	0
Test Defects Per unit size.	20.64897								
Yield	84.44444								
Productivity	19.29791								
Cost of Quality - Appraisal	18.53257								
Cost of Quality - Failure	19.63947								
Cost of Qualiry - Appraisal - Failure Ratio	0.943639								
Defects Per Hour - Design Review.	4								
Defects Per Hour - Code Review.	16.3142								
Defects Per Hour - Compile.	1.98895								
Defects Per Hour - Test.	2.863636								
DRL: Design Review - Test.	1.396825								
DRL: Code Review - Test.	5.697022								
DRL: Compile - Test.	0.694554								

Table 18. PSP Career Report.

PSP Project Plan Summary									
Name:	Couprie		Instructor:	Maurer					
Project Number:	Standard		Language	JAVA					
Project Name	1A		Date	SE0898					
			Phase	Time in Phase		Defects Injected		Defects Removed	
User Metric Name	Estimated Value	Actual Value		Plan	Actual	Plan	Actual	Plan	Actual
			TOTAL	204	234	6	8	6	8
Base	58	58	Plan	15	19				
Deleted	31	24	Design	31	39	1	2		
Added	43	57	DR	16	30			1	1
Reused	48	48	Code	50	41	5	6		1
Modified	19	24	CR	25	21			5	4
New and Changed	62	81	Compile	10	7				1
Total	118	139	Test	27	40				1
Defect Density	96.77419355	98.7654321	Post	30	37				
Test Defect Density	0	12.34567901							
Yield	100	87.5							
Productivity	18.23529412	20.76923077							
COQ - Appraisal	20.09803922	51							
COQ - Failure	18.1372549	47							
COQ - A-F Ratio	1.108108108	1.085106383							
Defects Removed in Design Review	3.75	2							
Defects Removed in Code Review	12	11.42857143							
Defects Removed in Compile	0	8.571428571							
Defects Removed in Test	0	1.5							
DRL - DR-Test	#DIV/0!	1.333333333							
DRL - CR-Test	#DIV/0!	1.333333333							
DRL - Compile-Test	#DIV/0!	5.714285714							

Table 19. PSP Project Plan Summary.

IMAGE EVALUATION
TEST TARGET (QA-3)