ANALYSIS OF PSP-LIKE PROCESSES FOR SOFTWARE ENGINEERING

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Paul Jefferson Conrad

June 2006

ANALYSIS OF PSP-LIKE PROCESSES FOR SOFTWARE ENGINEERING

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

by

Paul Jefferson Conrad

June 2006

Approved by:

Dr. Richard. Botting, Chair, Computer Science	Date
Dr. Ernesto Gomez	
Dr. Keith Schubert	

ABSTRACT

The PSP, Personal Software Process, is introduced to Computer Science graduate students in Software Engineering (CSCI655). The purpose of introducing PSP to Computer Science students is to allow students to enhance their coding skills and documentation. The PSP requires the software developer to record information about the source code. The gathered information is analyzed through various statistical techniques to help improve the development skills of the software developer. The analysis is used as a tool to estimate future software projects and to help make software development better.

PSP is the leading approach for software developers to improve their own software development skills. However, the PSP data collection process is a time consuming task and error prone. This thesis will try to solve this problem with PSP. The purpose of this thesis is to provide the California State University, San Bernardino Department of Computer Science with an analysis and recommended solution to improving the software development process of graduating Computer Science students.

ACKNOWLEDGMENTS

I would like to thank Dr. Richard Botting for devoting his time being my advisor and wonderful mentor over many years. I would also like to thank Drs. Ernesto Gomez and Keith Schubert for taking the time out of their already busy schedules to serve on my committee. I want to thank my wife, Colleen, and daughters Abigail and Marion, for their support, patience, and encouragement while I was preparing myself academically to produce this thesis. I want to also thank my mother, father, and brothers in their support.

I want to thank Dr. Arturo Concepcion for putting together a Computer Science department that enabled me to grow into the computer scientist that I am. I want to also thank the numerous professors who taught me various topics in Computer Science, for their time and sharing their knowledge inside and outside the classroom. I also want to thank Dr. Josephine Mendoza for her guidance through the Computer Science Graduate Program. I would also like to thank Amy Niessen and Monica Gonzales for all of their assistance while I was a graduate student.

DEDICATION

To Colleen, Abigail, and Marion

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

BACKGROUND

1.1 Introduction

The scope of this thesis is to provide an analysis of the Personal Software Process (PSP)[6] and other PSP-like methodologies for the Department of Computer Science to get instructional tools to assist graduating high quality software developers. This thesis will investigate the PSP and other similar process improvement models used by software engineers and Computer Science students. The PSP is one of the leading detailed process models for measuring and improving the software development process. Other software process improvement models will be reviewed, as a part of the literature survey in this thesis and the results will be reported.

1.2 Purpose of the Study

The purpose of the study is to provide the Computer Science department with an analysis and recommended solution to improving the software development process of graduating Computer Science students. In this thesis, there are five deliverables to be produced. The first is the review of PSP related literature and other software process improvement related literature, and summarize the

readings. Secondly, interview the Computer Science faculty on PSP, software process, and summarize findings from these interviews. The third deliverable is to analyze the current student software development process that is being used in the Computer Science department. The fourth deliverable is to derive solutions for the Computer Science department to better educate Computer Science students and recommend a solution.

1.3 What Problem is Faced

With the computer industry becoming increasingly more competitive, it is important to properly educate Computer Science students in the preparation for such a competitive industry. A challenge for any Computer Science program is to help train students to become a high quality engineer. This thesis will be looking at problems with software process improvement methodologies and the solutions to these problems. The PSP is one of the leading approaches for software developers and Computer Science students to improve their software development skills. This thesis shows that the Personal Software Process (PSP) data collection process is a time consuming task and error prone. This thesis will try to solve this problem with a PSP-like approach.

The Computer Science department offers six core programming courses. These courses are Computer Science I (CSCI201), Computer Science II (CSCI202), Data Structures (CSCI330), Software Engineering (CSCI455), Foundations of Software Systems (CSCI599), and Software Engineering Concepts (CSCI655). The first four core programming courses are taught at the undergraduate level, and the remaining two core programming courses are taught at the graduate level.

In Computer Science I (CSCI201), the course covers concepts of computer software design, implementation, methods and environments using a current high-level language. The course also surveys computers, applications and other areas of Computer Science. In Computer Science II (CSCI202), the students perform analysis of problems and the formulation, documentation and implementation of their solutions. The students are also given the introduction to data structures with abstract data types. Lastly, students are introduced to software engineering principles for both individual and group projects. When the students take Data Structures (CSCI330), they are formally introduced to abstract data structures such as lists, stacks, queues and trees. Students are introduced to storage allocation and associated application

algorithms for the abstract data structures introduced in the course. In Software Engineering (CSCI455), Computer Science students are formally introduced to advanced techniques and technology used to produce large software systems. The course laboratory works with a software development environment that mimics a large software team working on a large-scale software development project.

Graduate students who have not been introduced to CSCI201, CSCI202, and CSCI330 at the undergraduate level are required to take these courses before taking Foundations of Software Systems (CSCI599). In this course, the graduate student is introduced to software development process that includes software life cycles, software techniques and technologies used to produce large software systems. This course is a refresher or catch-up course that covers the same topic areas as CSCI455 and Operating Systems (CSCI460) courses. The graduate students taking the Software Engineering Concepts (CSCI655) are formally introduced to the analysis of software requirements definitions, software systems design, implementation issues, verification and validation, and software maintenance techniques. The graduate student is also taught rapid prototyping procedures, operational and transformational paradigms of software development. The

graduate student is introduced to software engineering models and CASE tools that include reverse engineering and module reusability concepts. The graduate student is also taught applications in object-oriented programming languages. Figure 1 is a flow diagram of these core programming courses offered by the Computer Science department.

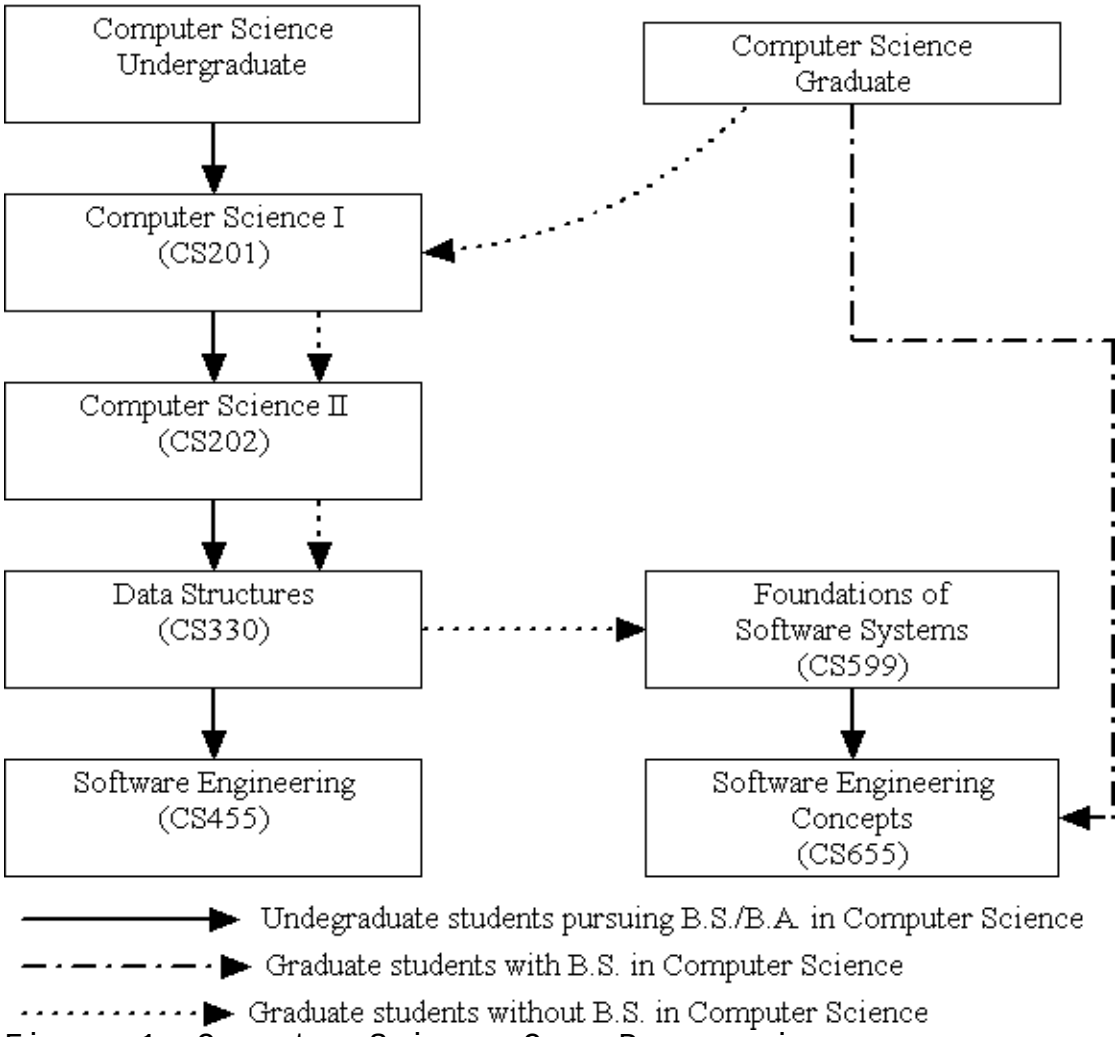

Figure 1. Computer Science Core Programming.

Table 1 contains the student enrollment for the CSCI201, CSCI202, and CSCI330 courses over the past six years. These enrollment figures are used in the scatter plots in figures three and four.

Table 1. Enrollment Figures for Past 6 Years.

Filled Seats in Courses			
Term	**CSCI201**	**CSCI202**	**CSCI330**
Fall 00	58	19	38
Winter 01	84	41	0
Spring 01	50	58	49
Fall 01	110	0	37
Winter 02	93	56	16
Spring 02	59	32	44
Fall 02	80	29	28
Winter 03	60	39	27
Spring 03	61	32	22
Fall 03	65	33	33
Winter 04	57	24	22
Spring 04	29	18	17
Fall 04	59	19	19
Winter 05	51	16	24
Spring 05	44	27	19
Fall 05	71	24	12
Winter 06	45	35	28
Spring 06	50	29	23

Figure 2 contains the student enrollment for the CSCI201, CSCI202, and CSCI330 courses over a period of the

past six years. The enrollment trends for these three courses have been a steady slow increase over the more recent terms.

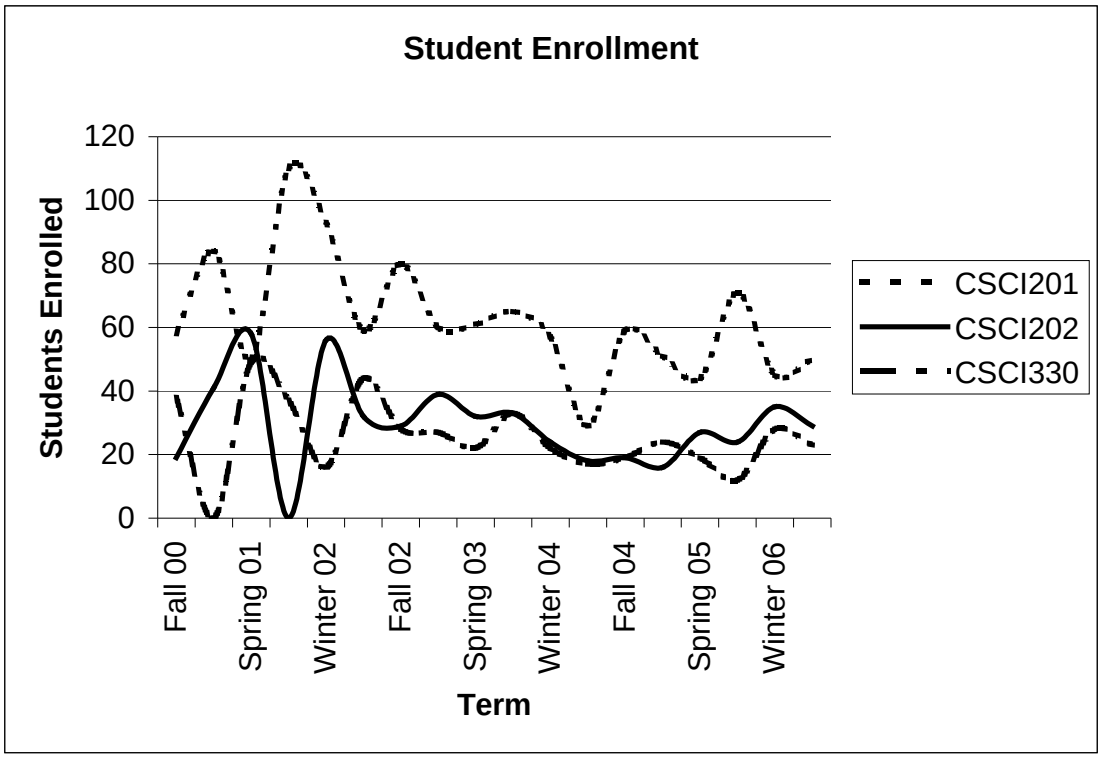

Figure 2. Student Quarterly Enrollments.

The scatter plot in Figure 3 shows the comparison between the CSCI201 and the CSCI202 enrollments for the term following the CSCI201 enrollment. The trend line shows that about half of the students who took CSCI201 continue with CSCI202 in the following term. The other half of the

students who do not take CSCI202 in the following term are most likely other College of Natural Sciences (CNS) majors such as Mathematics or Physics. The trend line also shows that about 3 students on average drop the CSCI202 course.

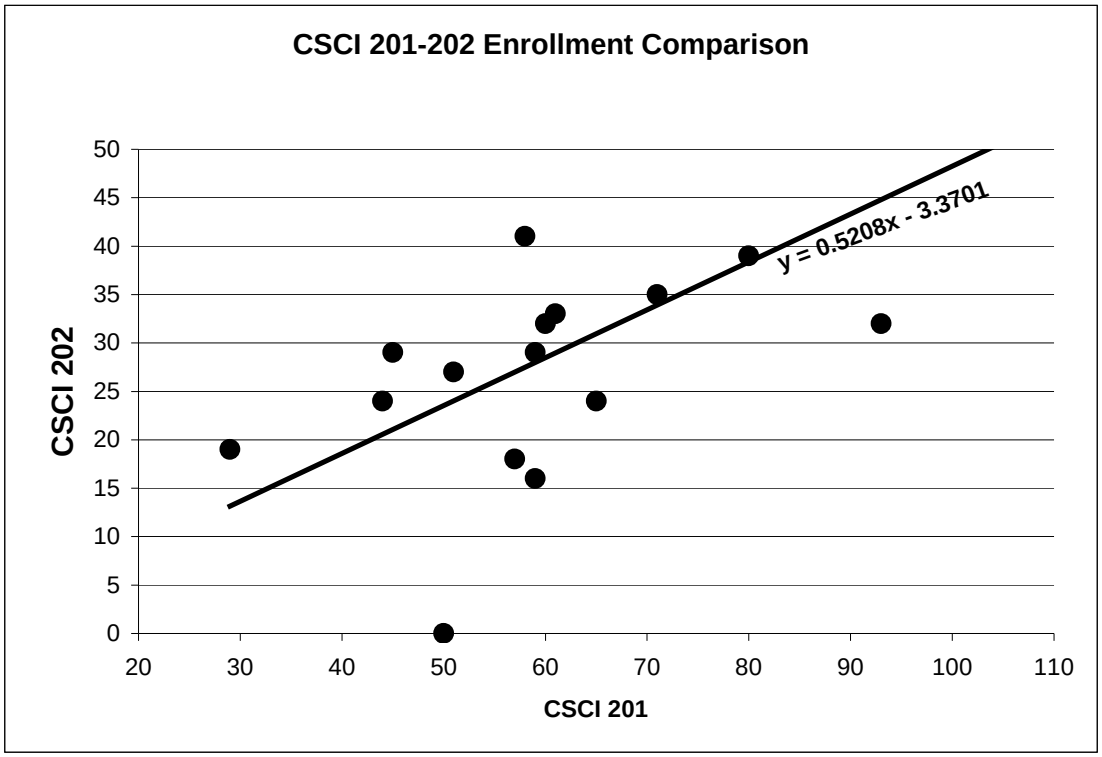

Figure 3. Computer Science I and Computer Science II Enrollment Comparisons.

The CSCI202/CSCI330 comparison scatter plot in Figure 4 does not show nearly as clear picture of the student behavior after taking the CSCI202 course. It could be

that students turn their focus towards the hardware core courses such as Digital Logic (CSCI310) and Machine Organization (CSCI313). The students could also be taking Programming Languages (CSCI320) or General Education courses if they are an undergraduate.

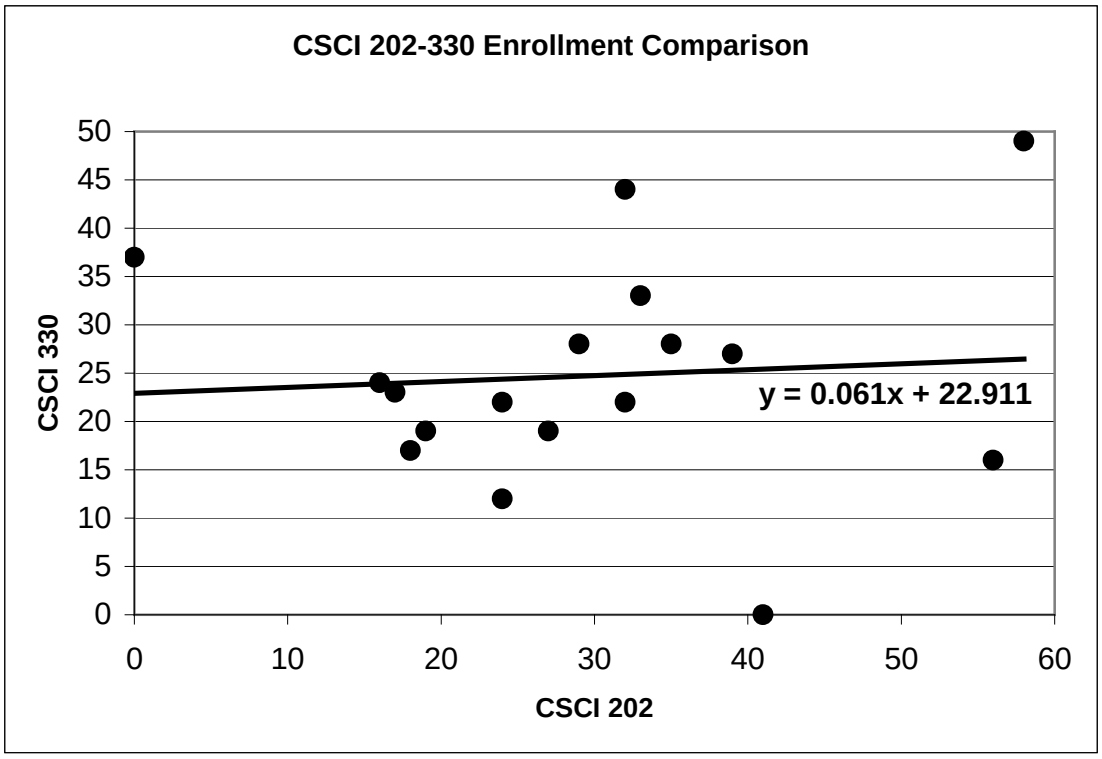

Figure 4. Computer Science II and Data Structures Enrollment Comparisons.

1.4 Significance for Study

This study should help the department significantly improve the number of high quality Computer Science graduates. With the generation of high quality Computer Science graduates, the industry is thus given a pool of strong talent, and the university can increase the likelihood of student body growth.

1.5 Assumptions

In this thesis, the derived solutions have to work within the existing course layout. This thesis is going to work at finding better tools to help aid the students taking the courses. Redesigning the existing Computer Science curriculum is not an option since it would be costly in terms of time resources.

1.6 Limitations

During this thesis, only the Computer Science faculty was interviewed. Student interviewing was not necessary since expert knowledge of software development practices was required to derive solutions. During the faculty interviews, general student opinions in the form of faculty feedback were noted and taken into consideration for the solutions. There was not any user testing done

since the PSP is introduced in the Software Engineering Concepts (CSCI655) course.

1.7 Definition of Terms

Throughout this thesis, there are several important terms that require attention and the definitions that should be noted.

Table 2. Terms.

Term	Definition
Actual work	The actual developer efforts devoted to a software development project (PSP).
Analyzed work	The calculated developer efforts devoted to a software development project (PSP)
Automated PSP	In which some or all of the derived measures are calculated and placed into the forms automatically.
C++	A hybrid, high-level programming language with object oriented features.
Capability maturity model	A methodology used to develop and refine software development process in an organization.
CASE	Computer Aided Software Engineering
Core programming course	Undergraduate courses, covering computer programming and data structures, which are required for fulfillment of a degree in Computer Science at CSUSB.
Cyclic development	Software development where there are refinements done in a cyclic manner (TSP).
Eclipse	An open source community with projects focused on providing an extensible development platform and application frameworks for building software.
Framework	A structure for supporting or enclosing something else.
Insights about work	Feedback for improving future software development activities (PSP).

Iterative	A process that goes through a series of approximations towards the optimal or correct solution. Each iteration repeats a similar series of activities. In software development, an iteration improves an existing piece of the software by adding new functionality. The iteration starts with planning, continues through analysis and design to testing.
Manual PSP	In which some or all of the derived measures are calculated and placed into the forms manually by hand.
Outcomes Assessment	An educational term naming a quality control process for courses and educational programs. At CSUSB, all degree programs are encouraged to have "Outcomes Assessment." The Computer Science department maintains a set of rubrics that define what students should have learned in the core courses and the actual outcomes are scored versus the rubrics each quarter.
Personal Software Process (PSP)	Software process framework showing software engineers how to manage project quality, make commitments, estimate and plan, and reduce defects.

Software subcontract management	A comprehensive set of tools to help plan and manage outsourced development projects, including a detailed process description, templates, checklists, and spreadsheet tools.
Team Software Process (TSP)	Software process framework showing software engineering teams how to manage product quality, make commitments, estimate and plan, reduce defects and effectively work in teams.
Unit test	Developer testing to demonstrate that a code module or unit meets its specified requirements.
Waterfall	Software life-cycle described by W. W. Royce where development is supposed to proceed linearly through the phases of requirements analysis, design, implementation, testing, integration and maintenance.

1.8 Thesis Organization

This thesis is divided into six chapters. Chapter One covers an introduction to the problem of the software process improvement, the purpose of this thesis, the significance of this thesis, encountered limitations, and definitions of terms that are used throughout this thesis. Chapter Two is a review of the literature covering the defining documents of PSP and the research that has been published about it. The pieces of literature that were reviewed for this thesis are relevant to software process improvement methodologies, and findings from various studies that were conducted were found to be helpful in

the analysis of methodology issues and concerns. In Chapter Two, a brief overview of the Personal Software Process (PSP) and the Team Software Process (TSP) is introduced. In Chapter Three, the methods of this thesis are introduced. Chapter Four presents the results from the thesis and looks into the results of interviewing Computer Science faculty as part of the deliverables for this thesis. In Chapter Five, the validation from the thesis is covered. In Chapter Six, this thesis lays out the roadmap for future research and development to be done in the area of providing software process improvement tools for Computer Science students. Finally, the references for this thesis are presented for guidance in future research work in this area.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

In order to take a closer look at what can be done with PSP to make it a valuable tool for students majoring in Computer Science, we have to investigate what areas of the PSP have educational values for the students. This chapter will describe the PSP based on the books by Watts Humphrey [6][9]. We also investigate how to reduce data quality errors made by students using PSP, examine ways to make administrative work less tedious, and explore other PSP-like methodologies that can be of high value for the Computer Science students.

2.2 Personal Software Process

Humphrey [7] explains that the PSP was developed by taking large-scale systems principles and applying these principles to small software development teams or organizations and individual software developers. PSP introduces making plans, managing the plans, reducing product defects, and increasing the software developer productivity. While many software developers feel that they already produce quality software, make accurate plans, and have high productivity measures, PSP provides

the framework and methods to help the developer gather supporting data about software quality, planning, and productivity. The supporting data obtained through the PSP allows the developer to build a case for management and customers in regards to software quality and the time estimates of the project.

Humphrey [6] lays out the details of the PSP. He introduces methods and practices in a gradual manner by using special training exercises. The maturity framework in the PSP is similar to those found in CMM (Capability Maturity Model). Figure 5, based on figure 1.2 (page 10) in [6], shows the relation between the PSP and CMM. The PSP does not include some aspects of CMM such as: software subcontract management and intergroup coordination, requirements management and software configuration management, and software quality assurance and training.

Software subcontract management and intergroup coordination cannot be applied to the PSP because they cannot be practiced at the individual level. The requirements management and software configuration management can be practiced at the individual level; however, a small team environment is the best place for these practices. When it comes to software quality

assurance and training, there is more of a relation to organizational issues and approaches with these aspects of CMM.

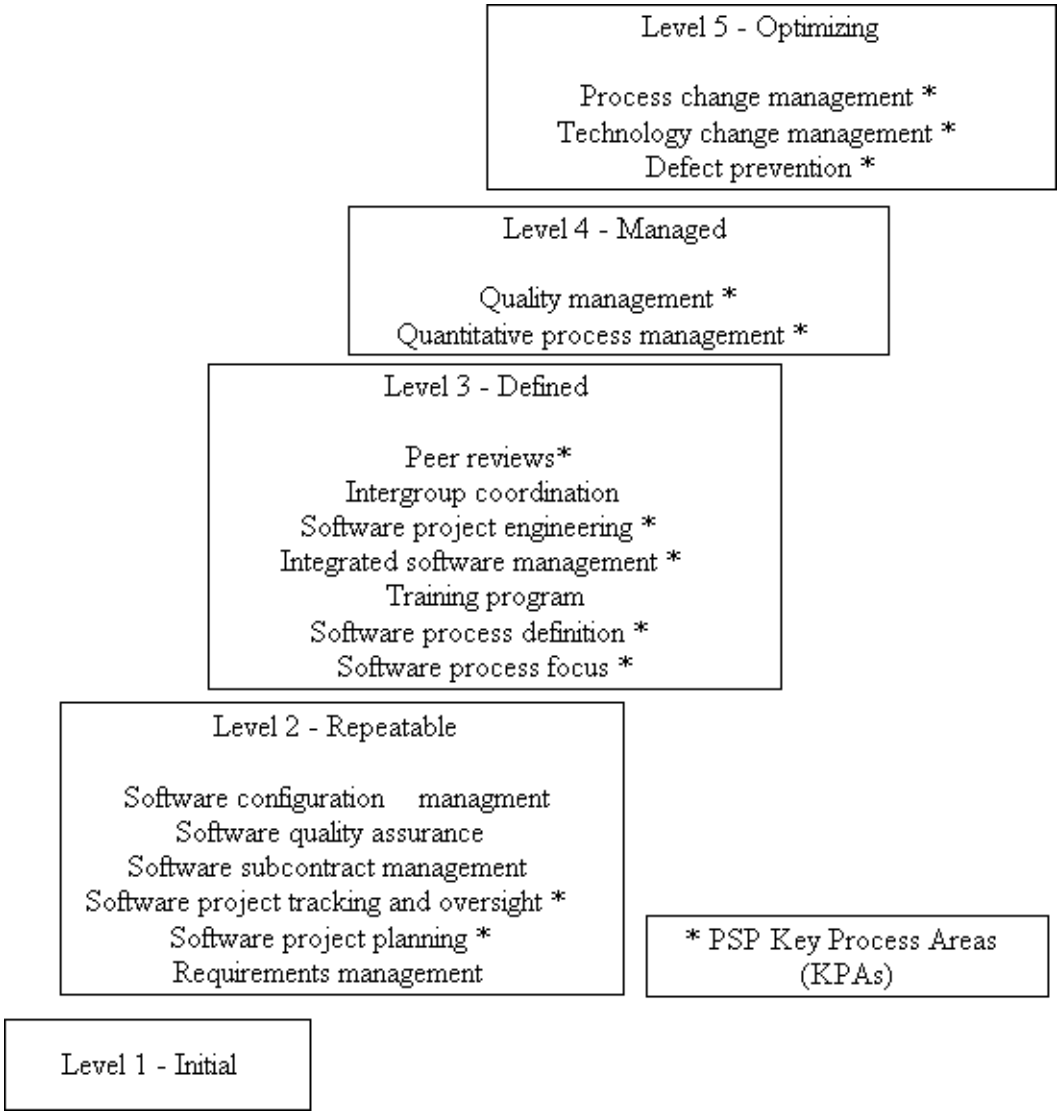

Figure 5. The PSP and CMM.

The PSP moves upward through five process levels starting with a baseline personal process, to the cyclic personal process level. In figure 6, based on figure 1.3 (page 11) in [6], the different stages of the PSP are laid out.

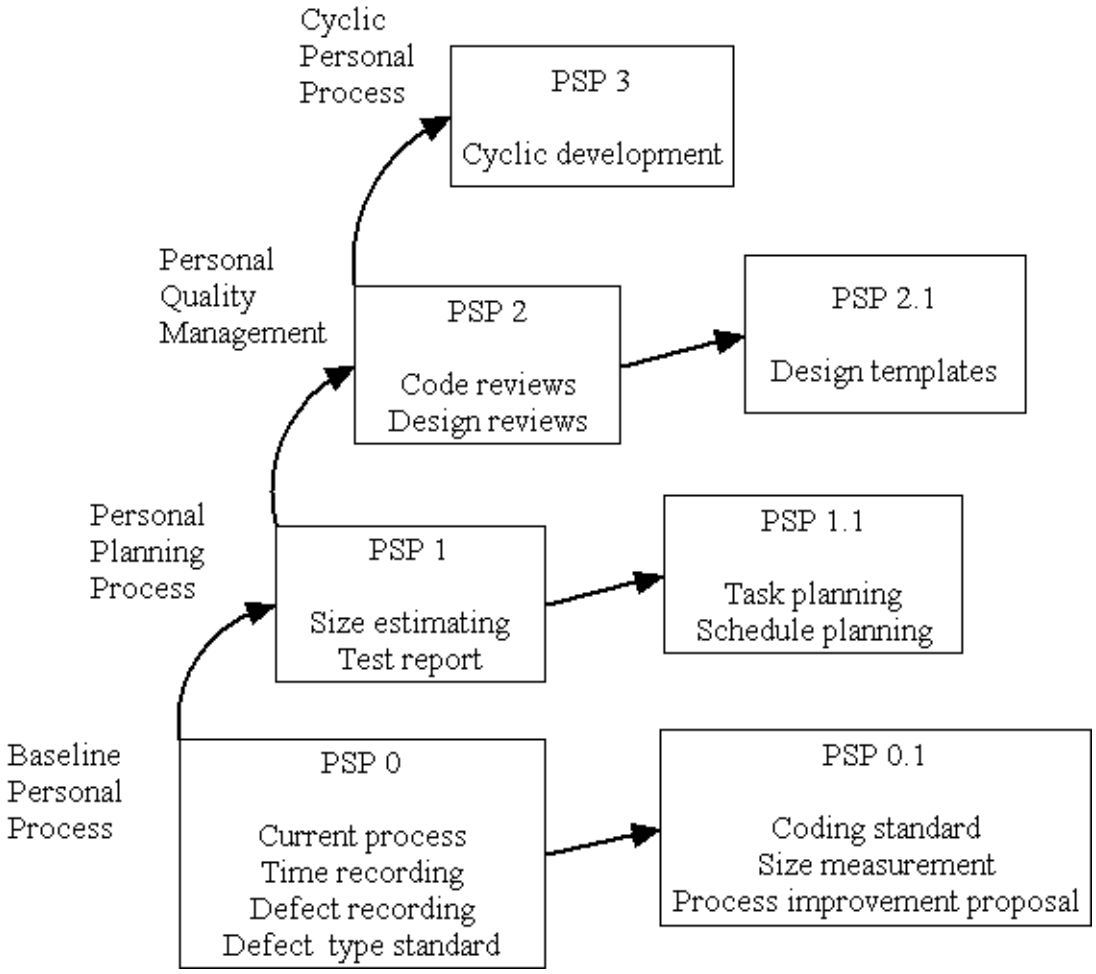

Figure 6. The Stages of PSP.

PSP starts at the lowest tier with methods in PSP 0. This stage is the starting point for all who learn PSP whether or not they are experienced programmers. PSP0 establishes the baseline personal process of the programmer. According to Humphrey [6], this stage provides the following: a structure that is well suited for handling small-scale tasks, a metric framework for measuring these small tasks, and a basic foundation for developing a process improvement approach. Two measures are recorded in the PSP0: the time spent per phase, and the number of defects found per phase. The phases used are the planning, design, code, compile, test, and postmortem. PSP0 also introduces time recording, defect recording, and a project plan summary. With time recording, the developer keeps track of how much time has been spent on each phase. The defect recording provides the developer with an approach at tracking which phase defects have occurred and which phase, if any, that the defect was removed from the program. The PSP0 is further enhanced with PSP0.1, which adds coding standards, size measurements, and process improvement proposals (PIP).

The next stage in the PSP is the PSP1 stage for developing personal planning process. In this stage, the same methods from PSP0 are used, but planning steps, size

estimation, and test reporting are added. Humphrey states that documented plans are needed: to help give an understanding of the relationship between the size of programs developed by the programmer and the time it takes to develop the program, to help put together commitments that can be met by the programmer, to have an orderly plan for doing the task, and to supply a framework for status reports of the task. There is the enhancement to PSP1, the PSP1.1, in which task planning and schedule planning is introduced.

PSP2 is the stage following PSP1. Here, the PSP introduces a personal quality management process to help track defects in the programs. The key aspect to this stage is to improve the quality of code written by the software developer. The first step to better code is using code and design reviews. PSP2.1 enhances PSP2 by utilizing design templates. It should be noted that design templates and design patterns are not the same. Design patterns are documented approaches to how future software is developed. In the PSP, the design template is a form that is used by the software developer for guidance through the current project.

The last stage of the PSP is PSP3 that concentrates on a cyclic personal process. Humphrey makes the point

that this stage allows PSP to scale upward efficiently to large programs. The main strategy in the PSP3 stage is to break a large program down into PSP2-sized pieces [6]. PSP3 relies on high quality increments of the task at hand.

Humphrey makes the assumption that if one collects data from past projects and compares the data with the current project, then one can supply good estimates of the effort involved. It is interesting to note that the PSP3 stage has been changed in 2005 to the Team Software Process (TSP) [9].

2.3 Team Software Process

The Team Software Process (TSP) is a defined framework for software teams [8]. TSP is geared for large projects that can span many years of development, and the TSPi is a scaled down version of the TSP. The TSPi is a design that modified TSP into an industrial process for software engineering teams of up to 20 developers. In the TSPi, the team starts with a small set of initial functionalities, and then through additional development cycles, learns to better plan and develop the product. This cyclic development strategy is comparable to

processes used by successful large-scale software development teams.

Humphrey has four basic principles in the TSPi. The first is that students learn most effectively when defined and repeatable steps are followed, and when rapid feedback on their process is available. The second principle Humphrey states that a defined team goal, an effective working environment, and capable coaching and leadership are the requirements for productive teamwork. Thirdly, students gain better understanding and appreciation for sound engineering practices when facing problems of realistic development projects and having guidance to effective solutions. The fourth principle is that, through effective instruction, learning builds on the available body of previous engineering, scientific, and pedagogical experience.

With these four principles, the TSPi design is based upon seven choices. The seven choices are: building a simple framework based on PSP, product development over several cycles, have an established standard for measures regarding quality and performance, have precise measures for teams and students, utilize role and team evaluations, have a sound process discipline, and also provide guidance for any teamwork problems.

Humphrey explores how and why the TSP and TSPi work through solutions for teamwork issues. He looks into why do projects fail, common teamwork problems, the definition of a team, what makes an effective team, how to develop an effective team, and how to use the TSPi to build an effective team. Humphrey first looks at why do software projects fail. He points out from DeMarco [DeMarco 88, pg 2] that it is not because of technical matters that projects fail, but rather because of teamwork problems. DeMarco says that

> "The success or failure of a project is seldom due to technical issues. You almost never find yourself asking 'has the state of the art advanced far enough so that this program can be written?' Of course it has. If the project does go down the tubes, it will be non-technical, human interaction problems that do it in. The team will fail to bind, or the developers will fail to gain rapport with the users, or people will fight interminably over meaningless methodological issues."

One key teamwork problem is handling job pressure. A tight job schedule or deadline is a common pressure that a team can be subjected to. Humphrey warns that effects of excessive pressure can be destructive since it can cause

team members to worry and conjure problems and difficulties that may not be an actual reality. The worries that come from pressure can often have untold consequences and potentially negative impact on the team.

Since pressure is a feeling that is generated internally by the developer, the first step in handling the job related pressure is to train developers to manage the pressure within themselves. The TSPi shows development teams how to manage pressure through a strategy and planning process. Since unrealistic schedules are a key source of software project problems, the TSPi helps software teams manage projects efficiently. When this happens, software teams have a better likelihood of performing quality work.

Humphrey investigates common team problems and shows how TSPi can address these problems. Ineffective leadership is a problem for teams and an effective leader is essential for a successful team. With an effective team leadership, teams can maintain focus on their plans and personal discipline. Sometimes one or more team members may not work together well, or work with the team well. This lack of cooperation and failure to compromise is a problem that does not arise often, but when it does, it must be handled in an effective and constructive way.

Humphrey states that peer pressure can remedy the problem, but if the problem continues, then instructor or manager interaction is needed to keep the team functioning.

Other team problems commonly found are: lack of participation, lack of confidence or procrastination, poor quality issues, function creep, and ineffective peer reviews or evaluations. With the lack of participation,

strengths and weaknesses are pointed out, then the opportunities to improve and motivate are possible.

Humphrey looks at what makes a team. He agrees with the definition of a team by Dyer [Dyer, pg 286]. In the definition, a team is basically two or more persons, who: work together towards a common goal, where each person is given a specific role or task, and the goal achievement is met by some form of a dependency amongst the members of the team. In order for a team to be successful, the ability to build an effective team is needed. Having team cohesion, goals, feedback, and a common working framework are necessary for building an effective team. Team cohesion is the act of making the team a tight-knit unit in which groups or members communicate freely and often. Humphrey states that friendship is not a necessity, but working close together with respect and mutual support is an essential requirement for team cohesion.

Challenging goals are an important element for building an effective team. Setting goals such as: detailed plans, performance targets, quality objectives, and schedule milestones, can give the team focus. When goals are tracked and progress is visibly displayed, the team members can see how the team is progressing towards

the final goal. Hopefully, the goals can serve as a positive motivation towards the final project goal.

Having a feedback mechanism is critically important for building an effective team. With a feedback process, the team members can gauge their performance. They can compare their performance to the team as a whole. At this point, their individual contribution to the team may or may not be apparent. Shirking team members are those team members who do not exert an equal amount of personal effort as the rest of the team. Feedback can help identify those shirking team members and address any issues that may be detrimental to the health of the project.

A common working framework serves as a pathway to achieving the team goals. This common working framework is the last element in building an effective team. Team members need to feel that the team tasks are achievable and they must know the following: what is the task to be do, when is the task to be done, what order of steps in the task are needed to complete the task, and who is going to be responsible for completing the task. By asking these questions, the team can have an effective plan showing where it is going towards the goal, and have an open channel for team communication.

Humphrey details how to develop effective teams and how the TSPi helps the team building process. The first step in the effective team building process is to create a jelled team. This jelling process is the convergence of team members to a common knowledge of what the product that is going to be built will do. Making plans and goals is the starting point of this process. Once this step is completed, the team members agree on the strategy and plan to build the product. The TSPi helps teams jell by providing steps for goal definitions, establishing team member roles, setting strategic approaches for achieving the set goals with plans, setting up a communication framework for teams, and external communication to the instructor or manager.

In order for all of this to happen, the TSPi is divided into eight major process scripts. These eight process scripts are: launch process, development strategy, development planning, design process, test plan, build process, system and integration testing, and the postmortem. In the launch process, the instructor or manager assigns and reviews team member assignment and roles. Also done in this launch process, is the setting of project objectives, and team and individual goals. The last step of the launch process is to review

the roles and working practices to make sure they will work to result in the finished project.

Development strategy and planning are the next two processes that make up the TSPi. With development strategy, the team can put together a strategy for doing the work and perform estimation on the sizes of the products and the required time to do the work. The development strategy needs to be documented so the team has a detailed roadmap showing where they need to go in the project. Once the development strategy is put into place, the development planning is done for the project. The development plans show how the project is going to be implemented. The development plans document what the requirements are, why the requirements are needed, and the key requirements issues are noted and the approaches to handling the issues are in place. All of this put together, the teams can be guided in a direction of doing better work.

The fourth process of the TSPi is the design process for the project. In the design process, the TSPi covers the design principles, team design practices, standards for design, design for testing, design for usability, and design inspection and design review practices. The design process also takes into

account the implementation process by starting with the design completion constraints, the implementation standards and strategies, and review and inspection of the implementation process. This is important for successful deployment of the finished project.

Humphrey explores the next four processes of the TSPi. These processes are: the project test plan, the build process, and the system and integration testing, and the postmortem. With the first three processes coupled together, the team can track down defect prone areas of the project. The last process, the postmortem process, the team members can learn from the work done in the project by reviewing team and individual work, examine what was done in each development cycle, and determine how the team can improve the next time.

The TSPi was summarized in this thesis because it is the next step above the PSP. Though students can learn the TSP and TSPi without any formal introduction to the PSP, the PSP is the cornerstone foundation for the TSP. Since in the computer industry developers often work on teams, the TSPi is a set of tools that can be beneficial for the student. Ideally, the student learns the PSP and advances to the TSP.

2.4 Software Engineering Teaching Tools

Martin Dick [3] looked at teaching tools that help aid software development training of Computer Science students. A positive learning environment depends on the development and integration of several teaching tools. No one particular tool can be the cure-all silver bullet for teaching, or be the only solution for Computer Science students. Computer Science students need to be involved in the Computer Science courses as active learners and not just passive receivers of information. When information is regurgitated back in the form of assignments, or examination results, it has not been properly digested.

The students could use PSP as an integral part of their software engineering training. The use of basic PSP measures in the software engineering coursework should allow the student to improve their software engineering and process improvement skills. The greatest amount of learning occurs when the assignment has relevance to the course and the student.

A survey of student attitudes toward assignments at Monash University [10] was done using a 1-5 Likert scale. Assignments were measured based on quantity of assigned work (1=little, 3=modest/okay, 5=excessive/too much),

student interest in assignment (1=high, 3=modest/okay, 5=little/dull), and learning value of assignment (1=high value, 3=moderate value, 5=little value). In Figure 7, based on figures 1, 2, and 3 from [3].

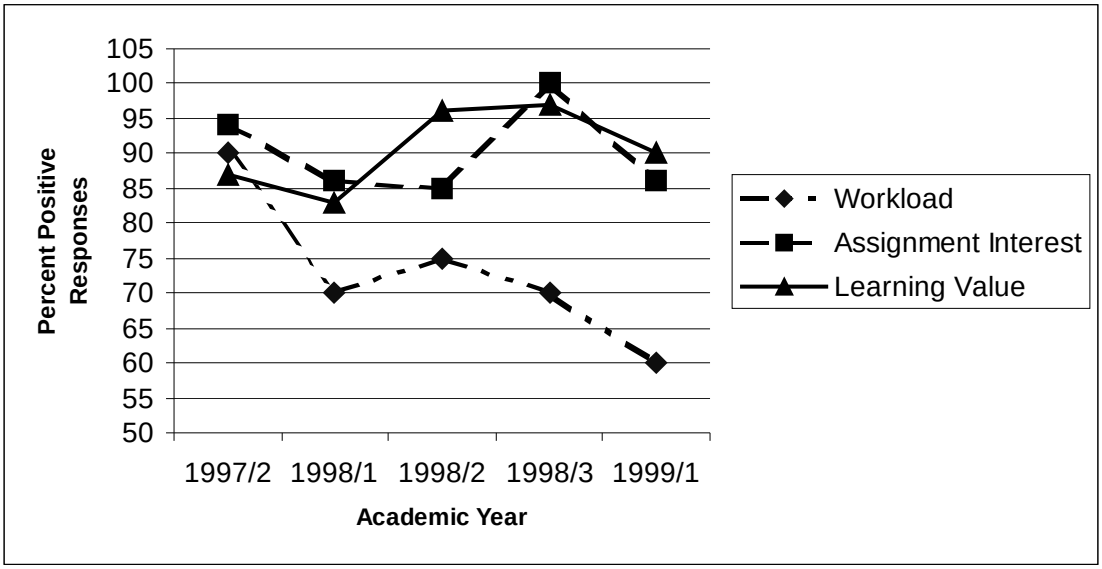

Figure 7. Student Responses.

Workload demonstrates the percent of students with a response of 4 or 5, pointing out that students believed the assignment workload to be too heavy. The response to the initial survey was to reduce the number of stages in work from 3 down to 2. In the area of assignment interest, the percent response levels were 1 to 3 for

interest in the assignments. Student interest has been in the area of being OK when the course was offered as a summer semester course; there was a strong interest during the 1998/3 term. This spike can be possibly attributed to the composition of students, who overall exhibited a higher level of interest and success in the course than in other semesters. The students undertaking summer semester were typically fast-tracking their degree and are therefore a more highly motivated group. Learning value gave the percentage of students who thought the assignments provided a challenge. It seems that there was reasonable interest since the assignments were perceived as an important contribution to the student knowledge of software engineering practice.

2.5 Personal Process Improvement Strategies

O'Connor [11] explores two personal process improvement methodologies and compares them. These two personal process improvement methodologies are PSP and the Process for Improving Programming Skills in Industry (PIPSI). Teachers, Computer Science students, and professional software engineers have found that learning PSP is both a demanding and challenging process. In PSP, there is significant investment in time and effort.

Reporting actual use of PSP in the software development industry has been limited due to several factors. First, many companies show reluctance to release data that may be used by customers or competitors to identify actual costs and defect levels. Secondly, little or no historical data that can help quantify effects on costs and schedule. Last of all, PSP has not been widely adopted in practice, resulting in fewer cases from which to draw conclusive results.

A significant benefit of PSP has been found in both the classroom and industry settings. In the classroom setting, Computer Science students reduce the number of defects in their program code, while on the same token, not impacting their productivity in a negative manner. In the industry setting, professional developers have improved both the accuracy in estimation and quality of the finished product.

However, problems are reported as a high rate of recidivism where PSP trained engineers do not maintain the disciplines taught and revert to their pre-PSP development processes. Other problems encountered are the duration of training involved being unsustainable for small and medium-sized enterprises, lack of tool support and data

recording via pencil-paper-spreadsheet is a tedious task for the developer.

PIPSI aims to provide a process improvement framework for software engineers in small or medium-sized enterprises, and to improve software engineering skills. There are three main deliverables with PIPSI: defining a personal process, personal project management, and personal quality management. O'Connor concludes that PIPSI is good tool for improving software process improvement since it takes out the administrative burden that is found in the PSP.

2.6 Data Quality Issues

Empirical software process improvement requires both gathering large amounts of data and the analysis of the data. Substantial effort is required for the data collection, analysis, interpretation of the information found in the analysis, and the introduction of organizational changes based on the found measurements. PSP is an "alternative and complementary" approach for which empirically guided software process is tailored to the individual software engineer.

Errors can affect the effectiveness of PSP. Errors can occur in data collection, and during the analysis of

the data. Disney and Johnson [4] devised two models of PSP in order to guide the way for an understanding of data quality problems that can arise in PSP. These models are labeled "Actual Work" and "Analyzed Work." In the "Actual Work" model, the developer collects primary measures for time, defects injected, and the work product characteristic, which Disney and Johnson refer to as "Records of Work." The "Analyzed Work" is the analysis of these collected primary measures. Disney and Johnson make the point about "Analyzed Work" helping yield "Insights about Work," which will guide the software developer in future software development activities.

PSP is done in two ways, "manual PSP" and partially or fully "automated" PSP. In the manual PSP, the software developer is responsible for entering measurements into forms by hand, editing an online version of the form, or filling out a printed copy of the form via pen or pencil. Disney and Johnson state that even spreadsheets can be considered as manual PSP. Unless the spreadsheet automatically inserts and maintains calculations, the values in the appropriate cells in the spreadsheet may be incorrect.

The partially or fully "automated" PSP is one in which some or all of the measures are calculated and

placed into the location on the form for the calculated measure. In automated PSP, the analysis tools and forms that represent the PSP reports need to be tightly integrated. Even though the "automated" PSP can automate all of the analysis calculations, the collection stage is still a manual stage.

Data quality in PSP can be affected in three basic areas during the data collection aspect of PSP: omission errors, addition errors, and transcription errors. The omission error is when the developer, either by accident or intentionally, fails to record one or more of the primary measures of time, defect, or the work of the product itself. Addition errors occur when the developer places "Records of Work" with data that does not reflect upon the actual practice. Transcription errors occur when the developer does intend to record the "Actual Work" done, but makes a clerical mistake during the collection process.

PSP can encounter data quality problems in the analysis stage of manual PSP. The three areas in which the data quality can be compromised are: omission errors, calculation errors, and transcription errors. Omission errors are errors that are encountered when the developer fails to perform a required analysis of the primary data.

Calculation errors occur when a developer attempts to perform an analysis and does so incorrectly. Disney and Johnson give an example of a developer using a regression based estimation approach when the historical data is uncorrelated and makes the predictions invalid. Transcription errors are a problem for data quality when the developer takes the results of the analysis and moves the computed information in places on the PSP report form where the data does not belong. Table 3, based on results found in [4] shows the types of data quality errors, the number of occurrences of the error, and percentage that the error makes up as a whole.

Table 3. Data Quality Errors Encountered with PSP.

Error Type	Occurrences	Percentage
Calculation	705	46%
Blank Field	275	18%
Information Transfer between Projects	212	14%
Entry	142	9%
Information Transfer within Project	99	6%
Impossible Values	90	6%
Process Sequence	16	1%
Total	1539	

Disney and Johnson found that the most commonly occurring error type when using the PSP were calculation errors. These errors could very well be just arithmetic mistakes that any normal human being produces. This error type was applied to any data field in which the values were used in calculations ranging from arithmetic operations or linear regression. The second most common error was the omission of required values. Information transfer between projects was the third type of error. Taking values from fields in one project and misplacing the values into another project would destroy its value. Disney states that it is almost impossible to determine where this type of error comes from. Disney and Johnson found that entry errors made up 9% of errors. These could be errors of misplacing digits in the fields and could also result from the software engineer or student not understanding the purpose of the field or from using an incorrect method when selecting the data. Information transfer within projects was an error that made up 6% of the PSP data quality errors. These errors are similar to the information transfer between projects, except the errors would occur when information was transferred from one form to another form within the project. Impossible

values were another type of error that Disney and Johnson found. This type of error occurred when two values were mutually exclusive. Common occurrences of this error was when there were overlapping time entries in the time logs, defect fix times for a particular phase, or phases occurring in a different order than stated in the defect recording log and time recording log. The last type of error was errors in which process sequence was not followed. This type of error occurs in the time recording log showing the software engineer or student moving back and forth between PSP phases rather than sequentially moving from one PSP phase to the next phase in an appropriate manner for the process.

Disney and Johnson investigated the effects of the PSP data errors since some errors can have a minor ripple effect on the calculations whereas other errors can have an enormous, if not devastating, impact on the calculations. Table 4 gives insight on the severity of errors in regards to the ripple effect on the PSP calculations. The error types are ranked in order from the least severe to the most severe.

Table 4. Errors Ordered by Severity.

Error Type	Occurrences	Percent
No impact on PSP data	104	7%
Single bad value, single form	674	44%
Multiple bad values, single form	197	13%
Multiple bad values, multiple forms, single project	41	3%
Multiple bad values, multiple forms, multiple projects	523	34%
Total	1539	

The errors that have no impact on the PSP data are errors such as missing header data, incorrect dates in the time recording log, and the filling in of fields for a more advanced process. Errors that result in a single bad value on a single form are the second type of error in severity. This level of errors is used when a significant field, which affects no other fields, was left blank or had an incorrect value. The third level of error severity consists of errors that result in multiple bad values on a single form in a single project. This level indicates that an incorrect or blank value was used to calculate values for one or more fields that are used in the single form. The fourth level of error severity consists of

errors that result in multiple bad values over multiple forms in a single project. This level indicates that either a blank or incorrect value was used in determining values for one or more fields on one or more forms in the same project. The most severe level of error resulted in multiple bad values on multiple forms over the course of multiple projects. Errors of this severity affected future projects by use of incorrect or blank values that were inherited from situations where errors resulting in multiple bad values on multiple forms through out all of the projects involved.

Disney and Johnson made several conclusions in the study. First, they feel the study indicated that there is a need to explicitly assess collection and analysis data errors from others in the PSP community. This study looked at two types of errors that can impact the effectiveness of PSP and this study enables the PSP community to devise an approach to minimize the data quality errors. Secondly, they feel that PSP does have a substantial educational value for software developers. Third, an integrated tool to support PSP is not something to be "merely helpful," but is a requirement to help the PSP obtain high analysis-stage data quality. Lastly, the questions raised by the study should be resolved; PSP data

should not be used to evaluate the effectiveness of the PSP itself.

2.7 Software Process Improvement Measurement

Paulish and Carleton [12] found that many software engineering organizations strive to improve their software development process. However, only a few know what the best approaches at improving the development process could be for their organization. Software process improvement is motivated as a result of external regulations, strong competition, and/or the call for increased profitability. The software engineer can address the later two through higher productivity.

The selection and successful implementation of software improvement process is dependant on many variables ranging from current software process maturity, organization skill sets, business and organizational issues such as cost, risk, and implementation time. The prediction of success is difficult due to external environmental variables such as staff skill sets, acceptance for implementing new process, training, and efficiency of the actual implementation of the software improvement process. The investment in training and

effort involved in a new software improvement process is often a considerable barrier for success.

For the case studies, two key variables were used in the selection of sites for the study. First, the site needed to have a large variety of application domains, organization size, and product complexity. Secondly, organizational dedication to software process improvement was a must. Two types of data, primary data and environmental data, were collected throughout the study from the selected sites. Primary data allowed for performance measures to be calculated determining how well the project progressed in development. The primary data that was collected were: defects found per phase, product size measured in terms of function points or lines of code, effort, schedule duration time, and schedule cycle time. Environmental data was collected to measure the development environment characteristics. The data collected ranged from staff size, staff turnover rate, software process maturity level, and staff morale.

2.8 Software Process Improvement in Practice

Coleman [2] states that software project success is generally determined by the project meeting the expectation of the users, being delivered in a timely

manner, and adhering to the budget constraints. Some of the large corporations make attempts to ensure success in their software projects by following the chosen software process improvement methods, such as Capability Maturity Model Integrated for Software (CMMI-SW) and the International Organisation for Standardisation (ISO) 9001.

Small to medium sized enterprises (SMEs), due to their size, are faced with particular challenges when developing software, and one of these challenges is choosing an appropriate software process improvement model. Coleman reports on which factors influence the structure of software process in Irish SMEs and examines why standard process models are rejected in favor of a tailored minimum.

The Software Engineering Institute (SEI) reported that between the end of 1997 to the end of 2002, despite the years of marketing and promoting software process improvement methods, the use of software process improvement models was relatively low. Newer process models, such as Personal Software Process (PSP) and Team Software Process (TSP) have emerged as software process improvement methods tailored towards SMEs. Large companies have charged the PSP and TSP methods as overly prescriptive and bureaucratic. On the other hand, SMEs

are thriving on the process being "good-enough" for their organizational needs.

Coleman interviewed 15 companies in the study and found a large range of SPI models being used. Interestingly, none of the companies were using models in a textbook manner, but instead, removed elements or added some proprietary element to the chosen model. In the interviewed companies, three were using Extreme Programming (XP) as the process model. Of these three companies, two used XP aggressively, whereas none of the companies used XP in the scope of the twelve principles that make up XP. Rational Unified Process (RUP) or some approximate variation of RUP had been used in seven of the fifteen companies. These seven companies used RUP in a tailored manner within a proprietary model. Two of these companies subsequently shelved RUP.

Stepanek [15] points out that there are a number of misconceptions about Rational Unified Process. RUP is not a process but instead is a toolkit for building processes. All of the roles, activities, and artifacts are tools in the toolkit. Only in rare cases every tool in the toolkit is used. Stepanek states that circumstances of critical, multi-year projects with hundreds of developers would make up this rare case. Stepanek also says that RUP cannot be

used directly as-is out of the box, and there is a requirement for tailoring to be done. As a note, the IEEE process standard [13][14] requires organizations to tailor their process.

The remaining five companies used either versions of the waterfall method or some type of iterative development approach as their software process model. There were various factors involved when choosing the software process improvement model the organization was going to use. The primary factors were: CTO (Chief Technology Officer)/Development Manager background, customer/application type, situation pre-process, size of the project or team, product/service model, and influence of key staff members. The main perception of process is the fear of added administrative overhead, and added work of gathering and upkeep of information. Coleman makes note of some quoted interviewees. SMEs face difficulty when implementing CMMI-SW or ISO 9001 due to cost constraints. As one company CTO stated:

> "We knew we had too much [process] when there was more administration being done than development. I think that product development is about being inventive and creative and new ideas coming forward and being developed quickly into something

mainstream. And when you don't see that happening I think that too much is being stifled.”

There must be caution on exactly how much process there must be in the software development process. Another interviewee expressed concern for the burden of the administrative overhead by putting it, as “from a making money perspective, you want every engineer to be working on billable work every time.” One engineer shows concern about not being able to spend quality time in producing code, but over engineering by saying:

> “I think a lot of commercial products out there are vastly over-engineered. I have learned that the hard way through Yourdon and drew diagrams for 2 years and didn't produce any code.”

Another engineer expressed dismay about software engineers having to do administrative work rather than actually working towards software development by stating:

> “One of the things I don't like with software companies I have worked for is the amount of programmers who end up doing admin work that they don't particularly want to do. And they tend to be the most senior guys who will deliver the most bang for buck in terms of coding.”

Another interviewed engineer showed concern for having to write software to be delivered within a schedule and the burden of having to work with a process on top of the schedule pressure by saying:

> “I'm an engineer. I've got to write this software and it has to be delivered in 3 weeks time and there is the pressure of delivering that. And if you add process in on top of that, unless people get into the habit of doing it on a day-to-day basis, where you really instill it as it will take you 10 minutes a day or 8 hours at the end of the project, and at the end of the project you won't remember what happened if you did. But so often, people were filling in time sheets and lists 6 weeks after the project had finished in order that the quality process could be seen to pass its audit.”

Many of these voiced concerns about the burden of having a process underscore the need for tools to automate the process. With tools that help automate the process and allow software developers to develop software, then software development organizations can have the potential for success.

The interviewed SMEs reject CMMI-SW and ISO 9001 because of cost requirements and the bureaucratic overhead

often associated with the adoption of either of these processes. Common phrases that are often encountered are: rigid, baggage, bureaucracy, buried in paper, forced into filling out forms, bulky, heavy, major drag factor, overkill, and there is no time for this. The list goes on with many variations of these phrases. The cost of administration and bureaucracy are costs that organizations of any size wish to minimize. Adding more process is often seen as adding more unnecessary bureaucracy. It is important to choose a SPI model that can suit an organization and not be a burden to the health of the organization.

2.9 Extreme Programming

Beck [1] introduced the practice of Extreme Programming (XP). In the early days of software process methods, there were the waterfall and iterative models. These models both required analysis, design, implementation, and then finally testing of the developed software. Long development cycles were very risky since they could not adapt to sudden changes in the software requirements. Shorter development cycles were Beck’s answer to the problem. The waterfall and iterative

methods began to address the issue of development cycle length.

Extreme Programming (XP) takes the conventional software process that is found in the waterfall and iterative method, and turns it on the side. Instead of planning, analyzing, and designing for the distant future, XP requires programmers to do many small iterations. Each iteration includes planning, analysis, design, coding, and testing the new user requirements. In table 5, Beck introduces the thirteen practices in XP.

Table 5. Practices of Extreme Programming.

Practice	**Definition**
Planning game	Customer makes decision about the scope and timing of releases based on estimates provided by the programmers. Programmers only focus on functionality demanded by the story requirements on 3X5 cards for the particular iteration.
Small releases	System is put into production within a few months, before solving the entire problem. New releases are made frequently such as daily or monthly.
Metaphor	The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.
Simple design	The design runs all of the tests, communicates everything the programmer needs to communicate, contains no duplication in code, and has fewest classes and methods.

Tests	Programmers write new unit tests before starting to code new requirements. The tests are collected and all tests must run correctly. The customer writes functional tests for the stories in iteration. The customer tests are run at the end of the iteration.
Refactoring	Evolution of the system design through transformation of the existing design that keeps all the tests running.
Pair programming	All production code written by two people together at one machine/screen/keyboard/mouse.
Continuous integration	New code is integrated with the system after no more than a few hours.
Collective ownership	Each programmer improves any code anywhere in the system at anytime when there is opportunity.
On-site customer	Customer who sits with the programming team at all times.
40-hour week	Idea of no one working more than 40 hours in one week. Any overtime is an indicator of deeper problems that need to be addressed.
Open workspace	Team works in large room with small cubicles. Pair programmers work on computers set up in center of room.
Just rules	By being part of Extreme Programming, team members must sign up to follow the rules.

XP starts by having the customer, or instructor in the programming course, write stories. Each story is a software requirement written on a 3X5 inch card. The programmer estimates the effort required for implementing each story and then the customer selects a collection of cards that can be done in the next iteration using the programmer estimation. This is the planning game. It defines the scope of the next iteration.

The customer thinks about their priorities and consults the programmer about the effort required. The programmers then take the selected stories and reduce them into smaller-grained tasks. The first decision is to be made about what the project could do and what it should do first. Beck considers the period before a project first goes into production as a dangerous time. It needs to be completed as quickly as possible. The iteration starts with the programmers writing additional unit tests and then changing the software so that it passes them and all previous tests. The changes are integrated into the system as they pass the tests.

Meanwhile the user develops acceptance tests that are applied at the end of the iteration. The iteration stops at the end of a period of time such as a single week. If any stories are incomplete they are returned to the planning game. It is always possible that the customer needs may have changed and made an unimplemented story of less value than other stories.

The last step in an iteration involves taking the running code and refactoring it. This improves the structure in a systematic way with out changing the behavior of the software. The coding activity is unusual in that it involves two programmers at one workstation

called pair programming. This provides instantaneous peer review of code.

It is interesting to point out that there are several similarities between the PSP and XP. The main focus of both methods is to strive for high quality code from the beginning of the project. In the PSP, the programmer keeps a history of how many defects are in the program code. Lowering the number of defects per thousands of lines of code is the first step towards improving code quality in the PSP. High quality code can be achieved in XP through the pair-programming practice. Because the pair-programming practice is in itself a code review as the story is being implemented in a programming language, it is a form of check-and-balance that helps keep the quality of the code high. As one developer is keying in the code, the other developer is on alert for any syntactical errors or logical errors.

The pair-programming practice also hopefully ensures that the developed code will conform to whatever coding standard that the software development organization has established. In the PSP, the software engineer has a coding standard form that is filled out. Another similar aspect between the PSP and XP is planning and estimation of the project. Though both are similar with planning and

estimation, the scope of the planning and estimation are different. The PSP does planning and estimation over the long term, whereas XP is concerned about the short term.

2.10 Summary

The literature review investigated several different pieces of literature. This literature review looked at how the PSP has evolved from the first Humphrey PSP [6] book to the more current [9] book as of this thesis writing. This literature review also took a quick overview of the Team Software Process (TSP) to see what areas there may be in the software process improvement methods after the PSP. In the literature review, issues regarding data quality and administrative overhead of the PSP are two troublesome areas that this thesis will look into. The literature review provided insight on the student opinion of the PSP when taught in a software engineering course. In the literature review, looking at the opinions by software engineers in the industry about the PSP were taken into consideration.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

This chapter covers the methodology used in this thesis. It explains how the Computer Science faculty interviews were planned. The methods used for recording the information are defined and how the information was interpreted.

3.2 Data Collection

This thesis looks into the thoughts of the Computer Science faculty and also into literature of software process improvement from the classroom perspective. The results from the literature have been summarized in Chapter Two of this thesis. Every faculty member who was interviewed had their thoughts or opinions counted. The main questions in the interviews concentrated on the interviewed faculty member’s thoughts or opinions with regards to software engineering. The Computer Science faculty interview questions were structured in such a way to make an attempt to unearth new ideas. The questionnaire began by profiling the interviewed faculty member by asking about what they are interested in, as well as what areas they may be actively researching. This

is not a question of qualifying or disqualifying the interviewee, but to put them at ease and to understand their background.

The Computer Science students were not interviewed. Students who are taking the CSCI201, CSCI202, and CSCI330 courses are just beginning their journey into the Computer Science discipline. These students would not have much knowledge in area of software engineering or in software process improvement methods. This thesis investigates how to improve the teaching of software process improvement methods like the PSP and requires expert, professional opinion. Student opinions should be sought in the future research on the effectiveness of these courses. Perhaps this should be part of the department “Outcomes Assessment” process.

3.3 Recording the Results

In Chapter Four, this thesis takes a closer look at the results from the Computer Science faculty interviews. Pencil and paper note-taking was used during the interviews along with voice recording when permitted by the interviewee. The interviews were recorded in order to gather any information that may have been missed during the note taking process. After the interviews were

completed, each answer was summarized. The common responses were taken note of, plus any responses that were different.

3.4 Interpreting the Results

This was not a statistical sample and so very little calculation was done beyond tabulating frequent responses. The aim was insight into expert opinions and to generate ideas. Individual ideas have more value than averages in this thesis. The results from the Computer Science faculty interviews were interpreted as the thoughts from the Computer Science faculty at California State University, San Bernardino. The results may reflect the same thoughts or opinions from faculty at other Computer Science departments. Since the thoughts or opinions from the reviewed literature very closely mirrored the Computer Science department faculty thoughts, thus results from the interviews can be seen as guiding light towards finding an optimal solution teaching Computer Science students the concepts of software process improvement methods in the Computer Science department.

3.5 Schedule

This thesis was scheduled with three key parts to be done. The first part of the schedule was to review

literature that looked into software process improvement and tools that could assist in training students to be better software engineers. The second part of the thesis schedule was to put together interview questions for interviewing the Computer Science department faculty members. Once it was finalized with what questions were going to be asked, the next part of the interview process was to contact all faculty members with an invitation to an interview. The third and final part of the schedule for this thesis was to put together all of the findings and present the Computer Science department with a solution to help improve the teaching of software engineering practices.

3.6 Summary

The collection of information from the literature review and the Computer Science department faculty interviews tried to discover ideas for improving the teaching of software process improvement methods to the Computer Science students. In Chapter Four, this thesis will detail the insightful findings of the faculty interviews, in Chapter Five, this thesis will detail several solutions, and in Chapter Six, the conclusions and

roadmap towards future research in this topic will be made available.

CHAPTER FOUR

RESULTS

4.1 Introduction

The Computer Science faculty was interviewed to find out their views of software process improvement and what they taught students about this area. The interviews uncovered several approaches to teaching software engineering, which will be discussed and compared in this chapter plus faculty opinions about software process improvement.

The department has 13 faculty and about 7 are involved in teaching programming and software engineering. These thirteen faculty members were invited by e-mail to take part in the survey. Seven of these faculty members responded positively to the e-mail and telephone follow-up. One faculty member asked to be left out because they were not involved in teaching the CSCI201 and CSCI202 courses. Of these thirteen faculty members, seven were interviewed thus covering 54% of the target population. Notice that because the department is small there is no question of getting a large sample. As a result, the opinions that were gathered must be taken with a grain of salt since they probably do not apply to other Computer

Science departments. Even so the interviews generated many interesting and useful insights in the teaching of software process improvement.

4.2 Computer Science Faculty Interviews

In the interviews, the faculty members were asked several key questions about their approaches towards teaching software engineering. Table 6 contains the questions that were presented to the interviewees. The Computer Science faculty interviewees were given these questions in preparation for the interview so they could have time to answer the questions in an accurate manner during the interviews.

Table 6. Interview Questions for Faculty.

Interview Questions
What is your area of research / specialty?
How important is ease of use in software process for students?
Is improving software quality an important aspect for students?
What software process, if any, do you encourage students to use?
What are your thoughts on PSP (Personal Software Process)?
Are there any other software process improvement methods worth investigating?

The first question asked the faculty member about their research area or specialty. This question was an important question in order to achieve a profile of the faculty member. The next two key questions looked into the importance of the ease of use for using software process improvement from a student stand-point, and whether or not a student should be focused on just learning the course material or if they should focus on improving the quality of software written in the course. The fourth question that was presented to the interviewee was looking the software development process that the faculty member encourages, if there was any process at all. The final two key questions asked in the interviews was about the faculty member's thoughts on the PSP and if there were any other software process improvement methods that they may recommend.

4.3 Interviewed Computer Science Faculty

The faculty is a bouillabaisse of many areas of research and specialty. The common areas of research and specialty were in the general areas of technological tools, software engineering, software process, and application development. The unique areas of research and specialty were in areas such as Internet programming,

enterprise application development, distributed computing, Expert Systems, numeric computation and robotics. After reviewing the similarities and differences between the interviewed faculty members, there some common ground was found, and also some different ideas. The interview results were interesting.

4.4 Software Process for Students

The first area of the interview investigated the approaches at introducing software process to the Computer Science students. When the Computer Science faculty members who had often taught the CSCI201, CSCI202, and CSCI330 courses were asked about the importance of the ease of use for Computer Science students using the software process improvement practices, most of the interviewed Computer Science faculty felt that it was important to have an easy to use software process when teaching the course materials. One Computer Science faculty member gave the suggestion of trying different approaches, which depended on the scope of the project/assignment. Another Computer Science faculty member was noted saying that keeping approaches in a simple manner was important. There should not be any reason for making the teaching process of a software

process improvement method difficult while the student is learning the course material alongside with the software process improvement method.

The Computer Science faculty was asked about their thoughts on the importance of learning how to write quality computer program code while learning the software engineering concepts. The Computer Science faculty who taught the CSCI201 and CSCI202 courses felt that learning how to write quality computer programs was an important aspect alongside the course materials. One faculty member stated that the CSCI201 course was generating good quality work while learning the materials in CSCI201, the CSCI330 students needed to improve. There may be a link here.

The interviewed faculty were given the opportunity during the interviews to voice their opinion about what software process that were encouraged during the presentation of the CSCI201, CSCI202, and CSCI330 courses. There many different processes that were encouraged in the courses are: iterative methods for software development, timed boxed approaches, students participation in requirements gathering for the course assignments, meeting strict deadlines for the assignments, the utilization of use case diagrams, class diagrams, test

first approaches, and other approaches of having some form of structure to complete the assignments.

4.5 Faculty Thoughts

Since this thesis investigates the PSP and PSP-like approaches, the interviewed Computer Science faculty were asked about their individual opinion about the PSP and any other software process improvement methods. The main opinion of the PSP was that it was too bureaucratic. Most of the interviewed Computer Science faculty felt that having some method of an integrated statistics tool could help ease the pain of bureaucracy involved in the PSP.

The Computer Science faculty had some interesting thoughts on other methods besides the PSP and PSP-like tools. Several members of the Computer Science faculty made the suggestion of introducing the Computer Science students to the principles of Extreme Programming (XP). The Computer Science faculty members expressed a strong interest in using Eclipse in the CSCI201, CSCI202, and CSCI330 courses. There was suggestion of being able to use the PSP and PSP-like methods alongside of some Integrated Development Environment (IDE) like Eclipse. Other Computer Science faculty members felt that using the Unified Modeling Language (UML), flow charts, and

configuration management, should be areas that could be investigated.

4.6 Student Processes

Several different tools and approaches used by Computer Science students have been noted while the students have been completing the necessary coursework for a Computer Science degree. The commonly used editors for coding the source code were editors that come with almost all distributions of the Linux operating system. These editors were programs such as: vi, emacs, and gedit. There had been occasions of students using Visual Studio in the department Windows laboratory. The Computer Science students use the gnu C/C++ compiler when building code in C++ and use Sun's Java SDK that comes installed in the labs when writing code in Java is required. Eclipse is now in the process of being introduced to students and this is good to help familiarize them with the ideas that come with working in IDEs that can have multiple source files in projects. However, it has been found that Eclipse runs slowly on the existing machines and the department System Administrator is planning on implementing an upgrade of the laboratory machines to address this issue.

4.7 Summary

The Computer Science faculty interviews were quite productive in this thesis investigation. Many common thoughts were expressed as to the importance of teaching software process improvement methods to Computer Science students. There were some different points that were brought up by some Computer Science faculty members that were also taken into consideration. The process of adding the Eclipse programming environment to the Computer Science department is going to be helpful.

CHAPTER FIVE

ALTERNATE SOLUTIONS

5.1 Introduction

In this chapter of the thesis, several ways to help improve teaching software process improvement to the Computer Science students will be discussed. Each will be taken in turn with the description of the solution, what is good about the solution, and what is not good about the solution.

5.2 Solution 0: Do Nothing

This solution is perhaps the simplest solution in this thesis. In this solution, there are no changes to be done with the teaching of the software process improvement methods to the computer Science students. In the CSCI201, CSCI202, and CSCI330 courses, various approaches of software process are taught to the Computer Science students.

On a positive note, the current methods of teaching software process to the Computer Science students are good. The Computer Science students are taught several different methodologies based on the results from the Computer Science faculty interviews. This can be good since it can open the student to various different methods

for developing software whether it is in an academic setting or in the Computer Science industry. Since the courses are already established in the Computer Science department, there is no preparation overhead of adding or modifying courses in the Computer Science curriculum.

On a negative note, this is not a good long-term solution for the Computer Science department. With this approach, the department runs the risk of becoming stagnant with old technological ideas and therefore students will not receive the benefits of a cutting-edge education. Since the Computer Science discipline and the industry is ever changing in an ever so rapid manner, it is important for the academic health of the Computer Science department to maintain cutting-edge knowledge and ideas. Keeping to this solution of not doing anything at all to change the way students are taught software process methodologies, is a risk that a Computer Science department cannot take while preparing students for a competitive career in the Computer Science industry.

5.3 Solution 1: Explore Other Methods

A solution for the Computer Science department would be to explore other methods for software process improvement. Since there are numerous different methods

that could be investigated, it can open up the department to a wider spectrum of ideas. Since Computer Science is an ever-changing field and new ideas appear on a regular basis, looking at other methods of software process may be a good idea.

The exploration of other software process improvement methods is not a solution that is practical for the Computer Science department. This exploration of other methods would cause the existing approaches to be possibly in limbo since there may not be any certainty that any one method would become the established method to teach. The exploration would also cause continual turbulence to the curriculum. This would cause the academic health of the department to decline with students either dropping out due to confusion or graduating without a solid learning of software engineering practices.

5.4 Solution 2: Integrate Automation Tools

The PSP is established as the software process improvement method to introduce to the graduate students taking CSCI655. Since Disney points out the trouble areas for data quality issues that arise with manual entry of the PSP metrics, a solution for the department could be to integrate tools to automate the PSP into the existing

curriculum. These tools would provide a means for the student to use the PSP to guide the way to producing higher quality code and understand how to plan and estimate their work. These automation tools can take in the form of application programs or shell scripts.

5.5 Why Solution 2 is Preferred

The solution of using integrated tools for the PSP is the preferred solution for the Computer Science department. The solution has no negative impact on the curriculum. These integrated tools can be introduced to the curriculum without changing the way the instructor teaches the courses. Since there would not be any changes to the existing curriculum, the utilization of integration tools is an efficient solution for the Computer Science department.

5.6 Summary

This thesis had to look at what solutions could help improve the software process of Computer Science students. This thesis was concerned with finding a solution that could have the least amount of impact in terms of changing the curriculum or any other costs that could have a negative impact.

CHAPTER SIX

CONCLUSION AND FUTURE RESEARCH

6.1 Conclusion

This thesis explored the PSP and presented a solution to help the Computer Science department educate Computer Science students how to become better software engineers. Having integrated tools to help gather the metrics used in the PSP can help introduce the PSP in a more productive and friendly manner. The most efficient approach is to add the integrated tools into the existing curriculum without making any unnecessary changes to the curriculum.

6.2 Future Research and Ideas

The process of putting the integration tools into place is an easy process. The tools can be put together by Computer Science students through both an Independent Study (CSCI595/CSCI695) or in the form of a Master's Project (CSCI698).

Heng-Jui Tsao presented a Master's Project to the faculty of the California State University, San Bernardino Computer Science department with the PSP Scriber [16]. It

will be to the advantage of the Computer Science department to integrate the PSP Scriber and any PSP integration tools that are incorporated with the curriculum. The combination of the PSP Scriber and PSP integration tools will reduce the administrative burden found with the PSP and reduce data quality errors created by the students. When the administrative burden on the student is reduced, the student is then placed in a learning environment that can foster stronger learning.

Since Eclipse is likely to become an added tool for Computer Science students, any future work on adding integration tools should be in the area of building PSP plug-ins for Eclipse. The Eclipse plug-ins are written in Java and the Eclipse website [5] contains many useful tutorials and articles about building plug-ins for the Eclipse IDE.

Student survey of how well the PSP integration tools will need to be conducted. After the investigation of how well the tools are working, the department can then assess any other directions that may need to be taken. This can be done as another Master's Thesis to help the Computer Science department in better educating future Computer Science students.

Hopefully, this thesis serves well as a guide for the Computer Science department in the journey of providing an excellence for Computer Science.

REFERENCES

[1] Kent Beck, “Embracing Change with Extreme Programming”, IEEE 1999, pgs 70-77

[2] Gerry Coleman, “An Empirical Study of Software Process in Practice”, IEEE 2005

[3] Martin Dick, Margot Postema, Jan Miller, “Teaching Tools for Software Engineering Education”, ITiCSE 2000

[4] Anne Disney, Philip Johnson, “Investigating Data Quality Problems in the PSP,” ACM SIGSOFT 1998, 11/98, pgs 143-153

[5] See http://www.eclipse.org/

[6] Watts S. Humphrey, “A Discipline for Software Engineering,” Addison-Wesley, 1995

[7] Watts S. Humphrey, “Why Should You Use A PSP,” ACM SIGSOFT 1995 Vol 20 Num 3, pgs 33-36

[8] Watts S. Humphrey, “Introduction to the Team Software Process”, Addison-Wesley, 2000

[9] Watts S. Humphrey, “PSP: A Self-Improvement Process fot Software Engineers”, Addison-Wesley, 2005

[10] See http://www.csse.monash.edu.au/

[11] Rory O’Connor, Gerry Coleman, “Strategies for Personal Process Improvement: A Comparison”, SAC 2002

[12] Daniel Paulish, Anita Carleton, “Case Studies of Software Process Improvement Measurement”, IEEE 1994, pgs 50-57

[13] “Standard for Information Technology – Software life cycle processes”, IEEE 3/1998, pg 1

[14] “Standard for Information Technology – Software life cycle processes – Implementation considerations”, IEEE 4/1998, pg 66

[15] George Stepanek, “Software Project Secrets: Why Software Projects Fail”, Apress 2005, pg 85

[16] Heng-Hui Tsao, “Personal Software Process (PSP) Scriber”, CSUSB 2002